Microsoft®

Microsoft® BASIC

Programmer's Guide

Version 7.0
For IBM® Personal Computers
and Compatibles

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Microsoft.

Copyright 1989, 1988, 1987 Microsoft Corp.
All rights reserved.

Simultaneously published in the U.S. and Canada.

Printed and bound in the United States of America.

Microsoft, MS, MS-DOS, CodeView, GW-BASIC, and XENIX are registered trademarks of Microsoft Corporation.

Betrieve is a registered trademark of Softcraft Inc., a Novell company.

Hayes is a registered trademark of Hayes Microcomputer Products, Inc.

Hercules is a registered trademark of Hercules Computer Technology.

IBM and PS/2 are registered trademarks of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

Olivetti is a registered trademark of Ing. C. Olivetti.

Times Roman is a registered trademark of Linotype AG and its subsidiaries.

Document No. DB0114-700-R00-1089

Table of Contents

Introduction xxvii

Part 1 Selected Programming Topics

Chapter 1 Control-Flow Structures 3

Changing Statement Execution Order **3**
Boolean Expressions **4**
Decision Structures **6**
Block IF...THEN...ELSE **8**
SELECT CASE **10**
Using the SELECT CASE Statement **12**
SELECT CASE Vs. ON...GOSUB **15**
Looping Structures **16**
FOR...NEXT Loops **16**
Exiting a FOR...NEXT Loop with EXIT FOR **20**
Suspending Program Execution with FOR...NEXT **21**
WHILE...WEND Loops **21**
DO...LOOP Loops **23**
Loop Tests: One Way to Exit DO...LOOP **27**
EXIT DO: An Alternative Way to Exit DO...LOOP **29**
Sample Applications **29**
Checkbook Balancing Program (CHECK.BAS) **29**
Statements Used **30**
Program Listing **30**
Carriage-Return/Line-Feed Filter (CRLF.BAS) **31**
Statements Used **31**
Program Listing **32**

Chapter 2 SUB and FUNCTION Procedures 35

Procedures: Building Blocks for Programming **35**
Comparing Procedures with Subroutines **36**
Comparing SUB with GOSUB **36**
Local and Global Variables **36**
Use in Multiple-Module Programs **37**
Operating on Different Sets of Variables **37**
Comparing FUNCTION with DEF FN **38**
Local and Global Variables **38**
Changing Variables Passed to the Procedure **38**
Calling the Procedure within Its Definition **39**

Use in Multiple-Module Programs 39
Defining Procedures 40
Calling Procedures 42
Calling a FUNCTION Procedure 42
Calling a SUB Procedure 43
Passing Arguments to Procedures 44
Parameters and Arguments 44
Passing Constants and Expressions 45
Passing Variables 47
Passing Simple Variables 47
Passing an Entire Array 47
Passing Individual Array Elements 48
Using Array-Bound Functions 49
Passing an Entire Record 49
Passing Individual Elements of a Record 50
Checking Arguments with DECLARE 50
When QBX Does Not Generate a DECLARE Statement 51
Developing Programs Outside the QBX Environment 52
Using Include Files for Declarations 53
Declaring Procedures in Quick Libraries 55
Passing Arguments by Reference 55
Passing Arguments by Value 56
Sharing Variables with SHARED 57
Sharing Variables with Specific Procedures in a Module 58
Sharing Variables with All Procedures in a Module 60
Sharing Variables with Other Modules 62
The Problem of Variable Aliases 64
Automatic and Static Variables 65
Preserving Values of Local Variables with STATIC 66
Recursive Procedures 67
The Factorial Function 67
Adjusting the Size of the Stack 68
Transferring Control to Another Program with CHAIN 69
Sample Application: Recursive Directory Search (WHEREIS.BAS) 72
Statements Used 72
Program Listing 72

Chapter 3 File and Device I/O 77

Printing Text on the Screen 77
Screen Rows and Columns 78
Displaying Text and Numbers with PRINT 78
Displaying Formatted Output with PRINT USING 80
Skipping Spaces and Advancing to a Specific Column 80
Changing the Number of Columns or Rows 81
Creating a Text Viewport 81

Getting Input from the Keyboard **84**
The INPUT Statement **84**
The LINE INPUT Statement **85**
The INPUT$ Function **86**
The INKEY$ Function **87**
Controlling the Text Cursor **87**
Positioning the Cursor **88**
Changing the Cursor's Shape **89**
Getting Information About the Cursor's Location **89**
Working with Data Files **90**
How Data Files Are Organized **90**
Sequential and Random-Access Files **91**
Opening a Data File **91**
File Numbers in BASIC **92**
Filenames in BASIC **93**
Closing a Data File **94**
Using Sequential Files **95**
Records in Sequential Files **95**
Putting Data in a New Sequential File **96**
Reading Data from a Sequential File **97**
Adding Data to a Sequential File **98**
Other Ways to Write Data to a Sequential File **98**
Other Ways to Read Data from a Sequential File **100**
Using Random-Access Files **102**
Records in Random-Access Files **103**
Adding Data to a Random-Access File **103**
Defining Records **103**
Opening the File and Specifying Record Length **104**
Entering Data and Storing the Record **105**
Putting It All Together **106**
Reading Data Sequentially **108**
Using Record Numbers to Retrieve Records **109**
Binary File I/O **110**
Comparing Binary Access and Random Access **110**
Positioning the File Pointer with SEEK **111**
Working with Devices **113**
Differences Between Device I/O and File I/O **114**
Communications Through the Serial Port **115**
Sample Applications **116**
Perpetual Calendar (CAL.BAS) **116**
Statements and Functions Used **117**
Program Listing **117**
Indexing a Random-Access File (INDEX.BAS) **121**
Statements and Functions Used **122**
Program Listing **122**
Terminal Emulator (TERMINAL.BAS) **130**

Statements and Functions Used 131
Program Listing 131

Chapter 4 String Processing 133

Strings Defined 133
Variable- and Fixed-Length Strings 135
Variable-Length Strings 135
Fixed-Length Strings 135
Combining Strings 136
Comparing Strings 137
Searching for Strings 138
Retrieving Parts of Strings 140
Retrieving Characters from the Left Side of a String 140
Retrieving Characters from the Right Side of a String 141
Retrieving Characters from Anywhere in a String 141
Generating Strings 142
Changing the Case of Letters 143
Strings and Numbers 143
Changing Strings 144
Sample Application: Converting a String to a Number (STRTONUM.BAS) 145
Functions Used 145
Program Listing 146

Chapter 5 Graphics 147

What You Need for Graphics Programs 147
Pixels and Screen Coordinates 148
Drawing Basic Shapes: Points, Lines, Boxes, 150
Plotting Points with PSET and PRESET 150
Drawing Lines and Boxes with LINE 151
Using the STEP Keyword 152
Drawing Boxes 153
Drawing Dotted Lines 155
Drawing Circles and Ellipses with CIRCLE 156
Drawing Circles 156
Drawing Ellipses 157
Drawing Arcs 158
Drawing Pie Shapes 161
Drawing Shapes to Proportion with the Aspect Ratio 162
Defining a Graphics Viewport 164
Redefining Viewport Coordinates with WINDOW 166
The Order of Coordinate Pairs 170
Keeping Track of View and Physical Coordinates 170
Using Colors 171
Selecting a Color for Graphics Output 172

Changing the Foreground or Background Color 172
Changing Colors with PALETTE and PALETTE USING 175
Painting Shapes 177
Painting with Colors 178
Painting with Patterns: Tiling 179
Pattern-Tile Size in Different Screen Modes 180
Creating a Single-Color Pattern in Screen Mode 2 181
Creating a Multicolor Pattern in Screen Mode 1 184
Creating a Multicolor Pattern in Screen Mode 8 187
DRAW: A Graphics Macro Language 189
Basic Animation Techniques 191
Saving Images with GET 192
Moving Images with PUT 194
Animation with GET and PUT 198
Animating with Screen Pages 203
Sample Applications 204
Bar-Graph Generator (BAR.BAS) 204
Statements Used 204
Program Listing 205
Color in a Figure Generated Mathematically (MANDEL.BAS) 210
Statements and Functions Used 210
Program Listing 211
Pattern Editor (EDPAT.BAS) 216
Statements Used 216
Program Listing 217

Chapter 6 Presentation Graphics 223

Presentation Graphics Program Structure 225
Terminology 227
Data Point 227
Data Series 227
Categories 228
Values 228
Pie Charts 228
Bar and Column Charts 228
Line Charts 229
Scatter Diagrams 229
Axes 229
Chart Windows 230
Data Windows 230
Chart Styles 230
Legends 231
Five Example Chart Programs 231
A Sample Data Set 232
Example: Pie Chart 232

Bar Chart 237
Line and Column Charts 239
Scatter Diagram 241
Customizing Presentation Graphics 244
Chart Environment 245
RegionType 246
TitleType 248
AxisType 249
LegendType 252
ChartEnvironment 253
Palettes 255
Colors 257
Line Styles 259
Fill Patterns 259
Plot Characters 261
Border Styles 261
An Overview of the Presentation Graphics Routines 261
Multi-Series Plots 262
Secondary Routines 265
Analysis Routines 265
Labelling Routines 265
Loading Graphics Fonts 266

Chapter 7 Programming with Modules 267

Why Use Modules? 267
Main Modules 268
Non-Main Modules and Procedure-Level Modules 270
Loading Modules 270
Using DECLARE with Multiple Modules 271
Accessing Variables from Two or More Modules 271
Using Modules During Program Development 272
Compiling and Linking Modules 272
Quick Libraries 273
Creating Quick Libraries 274
Tips for Good Programming with Modules 274

Chapter 8 Error Handling 275

Why Use Error Handling 275
How to Handle Errors 276
Setting the Error Trap 277
Writing an Error-Handling Routine 277
Exiting an Error-Handling Routine 278
Procedure- Vs. Module-Level Error Handling 281
Error Handling in Multiple Modules 285

Unanticipated Errors **286**
Guidelines for Complex Programs **292**
Errors Within Error- and Event-Handling Routines **292**
Delayed Error Handling **293**
Turning Off Error Handling **295**
Additional Applications **296**
Trapping User-Defined Errors **298**
Compiling from the Command Line **299**

Chapter 9 Event Handling 301

Event Trapping Vs. Polling **301**
How to Trap Events **302**
Where to Put the Event-Handling Routine **303**
Events that BASIC Can Trap **303**
Trapping Preassigned Keystrokes **304**
Trapping User-Defined Keystrokes **305**
Defining and Trapping a Non-Shifted Key **305**
Defining and Trapping a Shifted Key **305**
Trapping Music Events **308**
Trapping a User-Defined Event **310**
Generating Smaller, Faster Code **312**
Turning Off and Suspending Specific Event Traps **313**
Events Occurring Within Event-Handling Routines **315**
Event Trapping Across Modules **315**
Compiling Programs From the Command Line **316**

Chapter 10 Database Programming with ISAM 319

What Is ISAM? **319**
ISAM Statements and Procedures **320**
ISAM Vs. Other Types of File Access **321**
The ISAM Programming Model **322**
ISAM Concepts and Terms **323**
ISAM Components **328**
The ISAM Engine **328**
The Parts of the ISAM File **328**
ISAM File Allocation and Growth **329**
When to Use ISAM **329**
The Table/Index Model **329**
A Sample Database **332**
Designing the BookStock Table **332**
Creating, Opening, and Closing A Table **333**
Naming the Columns of the Table **333**
Specifying the Data Types of the Columns **333**
Data Type Coercion **334**

Opening the BookStock Table 335
Using OPEN and CLOSE with ISAM 335
Opening a Table 336
Closing a Table 336
The Attributes of filenumber% 336
Defining a Record Variable 337
Creating and Specifying Indexes on Table Columns 337
Indexes on BookStock's Columns 338
Creating a Unique Index 339
Subordering of Records Within an Indexed Column 339
Creating a Combined Index 340
Null Characters Within Indexed Strings in a Combined Index 340
Practical Considerations with Indexes 341
Restrictions on Indexing 341
Determining the Current Index 341
Transferring and Deleting Record Data 342
The Current Position 343
Changing the Current Index 343
Making a Different Record Current 343
Setting the Current Record by Position 343
Displaying the BookStock Table 344
A Typical ISAM Program 344
Setting the Current Record by Condition 354
Seeking on Strings and ISAM String Comparison 355
A Multi-Table Database 366
Deleting Indexes and Tables 374
ISAM Naming Convention 375
Starting ISAM for Use in QBX 375
Estimating Minimum ISAM Buffer Values 377
ISAM and Expanded Memory (EMS) 378
Using ISAM With Compiled Programs 379
Practical Considerations when Using EMS 381
TSRs and Installation/Deinstallation Order 381
Block Processing Using Transactions 382
Specifying a Transaction Block 382
The Transaction Log 383
Using Save Points 383
Maintaining Physical and Logical Data Integrity 386
Record Variables as Subsets of a Table's Columns 387
Using Multiple Files: "Relational" Databases 388
ISAM Utilities 389
ASCII Import/Export Utility (ISAMIO) 389
The ISAMCVT Utility 391
The Repair Utility 393
The ISAMPACK Utility 394

Converting Btrieve Code **395**
Run-Time Error Messages and Codes **399**

Chapter 11 Advanced String Storage 403

Far Strings Vs. Near Strings **403**
When to Use Far Strings **404**
Selecting Far Strings **404**
Direct Far-String Processing **404**
Calculating Far-String Memory Space **406**
Using Far-String Pointers **408**
Maximizing String Storage Space **410**
Far Strings and Older Versions of BASIC **413**
Data Structure and Space Allocation **414**

Chapter 12 Mixed-Language Programming 417

Organizing Mixed-Language Programs **418**
Mixed-Language Programming Elements **418**
- Making Mixed-Language Calls **419**
- Naming Convention Requirement **420**
- Calling-Convention Requirement **423**
- Effects of Calling Conventions **424**
  - Order in Which Arguments Are Pushed (BASIC, FORTRAN, Pascal) **424**
  - Order in Which Arguments Are Pushed (C) **424**
- Parameter-Passing Requirements **425**

Compiling and Linking **427**
- Compiling with Correct Memory Models **427**
- Linking with Language Libraries **428**

BASIC Calls to High-Level Languages **429**
- The BASIC Interface to Other Languages **429**
- The DECLARE Statement **429**
  - Using ALIAS **430**
  - Using the Parameter List **430**
- Alternative BASIC Interfaces **431**

BASIC Calls to C **431**
- Calling C from BASIC with No Return Value **432**
- Calling C from BASIC with a Function Call **434**

BASIC Calls to FORTRAN **435**
- Calling FORTRAN from BASIC—Subroutine Call **435**
- Calling FORTRAN from BASIC—Function Call **436**

BASIC Calls to Pascal **437**
- Calling Pascal from BASIC—Procedure Call **437**
- Calling Pascal from BASIC—Function Call **438**
- Restrictions on Calls from BASIC **439**
  - Memory Allocation **440**

Incompatible Functions 441
Allocating String Space 441
Performing I/O on BASIC Files 442
Events and Errors 442
Calls to BASIC from Other Languages 443
Other Language Interfaces to BASIC 443
Calling BASIC from C 444
Calling BASIC from FORTRAN 447
Calling BASIC from Pascal 449
Handling Data in Mixed-Language Programming 450
Default Naming and Calling Conventions 450
BASIC Conventions 450
FORTRAN Conventions 451
Pascal Conventions 451
Passing Data by Reference or Value 451
BASIC Arguments 451
Passing BASIC Arguments by Value 451
Passing BASIC Arguments by Near Reference 451
Passing BASIC Arguments by Far Reference 452
C Arguments 452
Passing C Arguments by Value 452
Passing C Arguments by Reference (Near or Far) 452
Effect of Memory Models on Size of Reference 453
FORTRAN Arguments 453
Passing FORTRAN Arguments by Value 453
Passing FORTRAN Arguments by Reference (Near or Far) 453
Use of Memory Models and FORTRAN Reference Parameters 453
Pascal Arguments 454
Passing Pascal Arguments by Near Reference 454
Passing Pascal Arguments by Far Reference 454
Numeric and String Data 454
Integer and Real Numbers 454
Strings 456
BASIC String Format 456
C String Format 457
FORTRAN String Format 458
Pascal String Format 458
Passing Strings Between BASIC and Other Languages 459
Passing Strings from BASIC 459
Passing BASIC Strings to C 460
Passing C Strings to BASIC 461
Passing BASIC Strings to FORTRAN 462
Passing FORTRAN Strings to BASIC 462
Passing BASIC Strings to Pascal 463
Passing Pascal Strings to BASIC 463
Special Data Types 463

Arrays **463**
Passing Arrays from BASIC **463**
Array Indexing and Declaration **466**
Declaring Arrays **467**
Compiling BASIC Modules for Row-Major Array Storage **467**
Array Data in Memory **468**
Passing Arrays Between Modules **468**
Arrays with More than Two Dimensions **469**
Structures, Records, and User-Defined Types **470**
External Data **471**
Pointers and Address Variables **471**
Common Blocks **472**
Using a Varying Number of Parameters **473**
B_OnExit Routine **474**
Assembly Language-to-BASIC Interface **475**
Writing the Assembly Language Procedure **476**
Setting Up the Procedure **476**
Entering the Procedure **477**
Allocating Local Data (Optional) **477**
Preserving Register Values **478**
Accessing Parameters **479**
Returning a Value (Optional) **481**
Numeric Return Values Other than 2- and 4-Byte Integers **482**
Exiting the Procedure **483**
Calls from BASIC **484**
Using CDECL in Calls from BASIC **486**
The Microsoft Segment Model **486**

Chapter 13 Mixed-Language Programming with Far Strings 489

Considerations When Using Strings **489**
String-Processing Routines **490**
Transferring String Data **490**
Deallocating String Data **492**
Computing String Data Addresses **492**
Computing String Data Length **493**
Passing Variable-Length Strings **493**
BASIC Calling MASM **493**
MASM Calling BASIC **498**
BASIC Calling C **501**
C Calling BASIC **503**
BASIC Calling FORTRAN **505**
FORTRAN Calling BASIC **508**
BASIC Calling Pascal **510**
Pascal Calling BASIC **512**
Passing Strings in QBX **515**

Passing Variable-Length String Arrays 515
Passing Fixed-Length Strings 517
Getting Online Help 518

Chapter 14 OS/2 Programming 519

Creating Real or Protected-Mode Programs 519
Editing Source Code 519
Language Changes for Protected Mode 520
    Protected-Mode-Only Statements and Functions 522
    Making OS/2 Calls in Your Program 523
        OS/2 Include Files 523
        Unsigned Values 523
        Pointers in User-Defined Types 524
        Far Character Pointers in Function Parameters 524
        Pointer to a Function in a Function Parameter 524
        Character in a Function Parameter 524
    Creating Dynamic-Link Libraries 526
    Creating Multiple Threads 526
    Making Memory References 526
    Using Graphics 527
    Using Music, Sound, and Devices 527
    Creating Extended Run-Time Modules 527
Compiling OS/2 Programs 527
Linking OS/2 Programs 528
    Using Standard Run-Time Modules and Libraries 529
    LINK Options for Real and Protected Modes 529
Debugging OS/2 Programs 529
Running BASIC Programs Under OS/2 530

Chapter 15 Optimizing Programs Size and Speed 531

Size and Capacity 531
BASIC Memory Use 531
    Variable-Length String Storage 534
    String Array Storage 535
    Numeric Array Storage 535
    Huge Dynamic Array Storage 536
Tips on Conserving Data Space 537
Controlling Program Size 538
    Using Stub Files 538
    Exploiting Run-Time Granularity with Code Style 541
        Avoid Accidental Use of Floating-Point Math 542
        Use Constants in SCREEN Statements 542
    Minimizing Generated Code 542
    Overlays 543

Minimizing Executable File Size 544
Compiling Programs for Speed 544
Compiling for the Target System 544
Math Options 544
80x87 Support 545
Alternate Math 545
Managing Data for Speed 546
Constants 546
Integers 546
Currency 547
Near Vs. Far Strings 547
Static Vs. Dynamic Arrays 547
Optimizing Control-Flow Structures for Speed 547
GOTO 548
GOSUB and DEF FN Subroutines 548
SUB and FUNCTION Procedure Calls 548
Writing Faster Loops 549
Optimizing Input and Output for Speed 550
Sequential File I/O 550
ISAM 550
Printing to Screen 550
Other Hints for Speed 551
Event Trapping 551
Line Labels 551
Buying Speed with Memory 551

Part 2 Using BASIC Utilities

Chapter 16 Compiling with BC 555

New Options and Features 555
Compiling with the BC Command 556
Specifying Filenames 557
Uppercase and Lowercase Letters 557
Filename Extensions 557
Paths 558
Using BC Command Options 558
Using Far Strings (/Fs) 561
Using Floating-Point Options (/FPa and /FPi) 561
In-Line Instructions (/FPi Option) 561
Alternate Math Library (/FPa Option) 562
Run-Time Modules and Floating-Point Methods 562
Optimizing Procedure Calls (/Ot) 562

Chapter 17 About Linking and Libraries 563

What Is a Library? 563
Types of Libraries (.LIB and .QLB) 564
Object-Module Libraries (.LIB) 564
Stand-Alone Libraries 565
BASIC Run-Time Module Libraries 566
Add-On Libraries 567
BASIC Toolbox Files 568
Quick Libraries (.QLB) 568
Advantages of Quick Libraries 568
Making Quick Libraries 568
Overview of Linking 570
Using LINK to Create Executable Files 570
Using LINK to Create Quick Libraries 570
Linking with Stub Files 571
Linking to Create Overlays 571

Chapter 18 Using LINK and LIB 573

Invoking and Using LIB 573
Old Library File 574
Consistency Check 575
Creating a Library File 575
Options 576
Ignoring Case of Symbols (/I) 576
No Extended Dictionary (/NOE) 576
Using Case-Sensitive Symbols (/NOI) 576
Specifying Page Size (/PA:number) 577
Commands 577
Order of Processing 578
How LIB Processes Commands 578
Add Command (+) 578
Delete Command (–) 579
Replace Command (– +) 579
Copy Command (*) 580
Move Command (–*) 580
Cross-Reference Listing File 581
New Library 581
LIB Prompts 581
Extending Lines 582
Default Responses 582
LIB Response File 582
Invoking and Using LINK 583
Default Filename Extensions 585
Choosing Defaults 586
LINK Options 586

Abbreviations **587**
Numeric Arguments **587**
LINK Environment Variable **587**
Valid LINK Options **587**
Invalid LINK Options **590**
Aligning Segment Data (/A:size) **590**
Running in Batch Mode (/BA) **591**
Preparing for Debugging (/CO) **591**
Ordering Segments (/DO) **591**
Packing Executable Files (/E) **592**
Optimizing Far Calls (/F) **592**
Viewing the Options List (/HE) **593**
Preparing for Incremental Linking (/INC) **593**
Displaying LINK-Process Information (/INF) **594**
Including Line Numbers in the Map File (/LI) **594**
Listing Public Symbols (/M) **595**
Ignoring Default Libraries (/NOD:filename) **595**
Ignoring Extended Dictionary (/NOE) **595**
Disabling Far-Call Optimization (/NOF) **595**
Preserving Case Sensitivity (/NOI) **596**
Suppressing the Sign-On Logo (/NOL) **596**
Ordering Segments Without Inserting Null Bytes (/NON) **596**
Disabling Segment Packing (/NOP) **596**
Setting the Overlay Interrupt (/O:number) **596**
Packing Contiguous Segments (/PAC:number) **596**
Packing Contiguous Data Segments (/PACKD) **597**
Padding Code Segments (/PADC:padsize) **598**
Padding Data Segments (/PADD:padsize) **598**
Pausing During Linking (/PAU) **598**
Specifying OS/2 Window Type (/PM:type) **599**
Creating a Quick Library (/Q) **600**
Setting Maximum Number of Segments (/SE:number) **600**
Issuing Fixup Warnings (/W) **600**
Object Files **600**
Executable File **601**
Map File **601**
Libraries **601**
- How LINK Searches for Libraries **602**
- Library Name with Path **602**
- Library Name Without Path **602**
- Searching Additional Libraries **602**
- Searching Different Locations for Libraries **602**
- Overriding Libraries Named in Object Files **602**

Module-Definition File **604**
LINK Prompts **604**
LINK Response File **605**

Options and Command Characters 605
Prompts 605
LINK Operation 606
Alignment of Segments 607
Frame Number 607
Order of Segments 607
Combined Segments 607
Groups 608
Fixups 608
LINK Memory Requirements 610
Linking Stub Files 610
Linking with Overlays 612
Using Expanded Memory 613
Restrictions on Overlays 613
Overlay-Manager Prompts 614

Chapter 19 Creating and Using Quick Libraries 615

The Supplied Library (QBX.QLB) 615
Files Needed to Create a Quick Library 616
Types of Source Files 616
The .QLB Filename Extension 617
Creating a Quick Library 617
Making a Quick Library from QBX 618
Starting QBX 618
Loading Desired Files 619
Unloading Unwanted Files 619
Creating a Quick Library 619
Additional Files Created 620
Making a Quick Library from the Command Line 620
Building a New Quick Library 621
Making a New Quick Library from BASIC Source Modules (QBX) 621
Making a New Quick Library from BASIC Source Modules (Command Line) 621
Making a Quick Library from Other-Language Modules (Command Line) 622
Making a Quick Library from BASIC and Other-Language Modules (QBX) 623
Making a Quick Library from .LIB Files (Command Line) 623
Making a Quick Library from Other Quick Libraries (Command Line) 623
Combining BASIC Source Modules and Existing Quick Library Modules (QBX) 624
Mouse, Menu, and Window Libraries 624
Loading and Viewing Quick Libraries 625
Loading a Quick Library 625
Using the /RUN Option 626
Quick Library Search Path 626

Viewing the Contents of a Quick Library **626**
Programming Considerations for Quick Libraries **627**
The B_OnExit Routine **627**
Handling Strings and Arrays **628**
Determining the Maximum Size for a Quick Library **628**
Making Compact Executable Files **629**

Chapter 20 Using NMAKE 631

NMAKE in a Nutshell **631**
Invoking NMAKE from the Command Line **631**
NMAKE Options **632**
Invoking NMAKE Using a File **634**
Description Files **635**
Description Blocks **635**
Wildcard Characters **637**
Escape Character **637**
Modifying Commands **638**
Specifying a Target in Multiple Description Blocks **639**
How Description Blocks Work **640**
Macros **640**
Macro Definitions **641**
Defining Macros on the NMAKE Command Line **641**
Defining Macros in Description Files **641**
Using Macros **641**
Macro Substitutions **642**
Predefined Macros **643**
Characters that Modify Predefined Macros **645**
Macro Substitutions on Predefined Macros **646**
Precedence of Macro Definitions **647**
Macro Inheritance **647**
Inference Rules **647**
Specifying Paths **649**
Predefined Inference Rules **649**
Directives **650**
Pseudotargets **652**
The TOOLS.INI File **654**
In-Line Files **654**
Keeping In-Line Files **655**
Using Whitespace Characters **656**
Creating More than One In-Line File **656**
Sequence of Operations **657**
Differences Between NMAKE and MAKE **658**

Chapter 21 Building Custom Run-Time Modules 661

Creating a Custom Run-Time Module **661**
- Compiling Source Files **662**
- Creating an Export List **662**
  - OBJECTS Directive **663**
  - EXPORTS Directive **663**
  - LIBRARIES Directive **663**
- Invoking the BUILDRTM Utility **663**
  - Building Default Run-Time Modules and Libraries **665**
  - Run-Time Messages from BUILDRTM **665**
  - Results from BUILDRTM **666**
- Creating Executable Files **666**

Programming Considerations for Custom Run-Time Modules **667**
- Handling Data Objects Consistently **667**
- Assembly Language Programs **667**
- DGROUP References **668**

Chapter 22 Customizing Online Help 669

What's in a Help File? **669**
- Contexts and Topic Text **669**
- Cross-References **670**
  - Implicit Cross-References **670**
  - Explicit Cross-References **670**
- Formatting Flags **670**

Help File Formats **671**
- QuickHelp **671**
- Rich Text Format **671**
- Minimally Formatted ASCII **672**
- Unformatted ASCII **672**

Invoking HELPMAKE **672**

HELPMAKE Options **673**
- Options for Encoding **673**
- Options for Decoding **675**

Creating a Help Database **677**
- Help Text Conventions **677**
  - Context Conventions **678**
  - Recommended Contexts **678**
  - Required Contexts **678**
  - Internal Help Context Prefixes **679**
- The Help Text File **679**
- Explicit Cross-References **680**
- Local Contexts **681**
- Application-Specific Control Characters **681**

Using Help Database Formats **682**
- QuickHelp Format **682**

The QuickHelp Context Command 683
The QuickHelp Comment Command 683
QuickHelp Formatting Flags 683
QuickHelp Cross-References 684
Minimally Formatted ASCII 686
RTF 687
HELPMAKE Error Messages 689

Appendix A Language Elements 695

Character Set 695
The BASIC Program Line 696
Line Identifiers 696
Line Numbers 697
Alphanumeric Line Labels 697
Executable and Nonexecutable Statements 698
Line Length 698

Appendix B Data Types, Constants, Variables, and Arrays 701

Data Types 701
Elementary Data Types — String 701
Variable-length Strings 701
Fixed-length Strings 701
Elementary Data Types — Numeric 702
Integer Numbers 702
Floating-Point Numbers 703
Currency Numbers 706
User-Defined Data Types 707
Data Types in ISAM Files 707
Constants 707
String Constants 708
Numeric Constants 708
Single-Precision Constants 711
Double-Precision Constants 711
Currency Constants 712
Symbolic Constants 712
Variables 712
Variable Names 714
ISAM Names 714
Declaring Variable Types 715
Type-Declaration Suffix 715
AS Declaration Statements 715
DEFtype Declaration Statements 715
Declaring Array Variables 717
Variable Storage and Memory Use 719

Variable-Length String Storage 719
String Array Storage 720
Numeric Array Storage 720
Huge Dynamic Array Storage 721
Scope of Variables and Constants 721
Global Variables and Constants 722
Local Variables and Constants 723
Sharing Variables 724
DEF FN Functions 724
Summary of Scope Rules 725
Static and Dynamic Arrays 725
Automatic and Static Variables 725
Type Conversion 728
Numeric Constants and Numeric Variables 728
Strings and Numerics 728
Type Conversion in Expression Evaluation 728
Type Conversion in Logical Expressions 729
Type Conversion Between Floating Point and Integer Values 729

Appendix C Operators and Expressions 731

Operators and Expressions Defined 731
Hierarchy of Operations 731
Arithmetic Operators 733
Integer Division 733
Modulo Arithmetic 733
Overflow and Division by Zero 734
Relational Operators 734
Logical Operators 735
Functional Operators 738
String Operators 739

Appendix D Programs and Modules 741

Modules 741
Procedures 741
FUNCTION Procedures 742
SUB Procedures 742
DEF FN Procedures 743
Passing Arguments to Procedures 743
Recursion 744
Overlays 745

Index

Tables

Table 1.1 Relational Operators Used in BASIC **4**
Table 1.2 Block IF...THEN...ELSE Syntax and Example **8**
Table 1.3 SELECT CASE Syntax and Example **12**
Table 1.4 FOR...NEXT Syntax and Example **17**
Table 1.5 WHILE...WEND Syntax and Example **22**
Table 1.6 DO...LOOP Syntax and Example: Test at the Beginning **24**
Table 1.7 DO...LOOP Syntax and Example: Test at the End **25**
Table 3.1 Devices Supported by BASIC for I/O **113**
Table 5.1 Color Palettes in Screen Mode 1 **173**
Table 5.2 Background Colors in Screen Mode 1 **173**
Table 5.3 Binary to Hexadecimal Conversion **184**
Table 5.4 The Effect of Different Action Options in Screen Mode 2 **195**
Table 5.5 The Effect of Different Action Options on Color in Screen Mode 1 (Palette 1) **197**
Table 6.1 Presentation Graphics Toolbox Files **224**
Table 6.2 Presentation Graphics Chart Styles **230**
Table 6.3 Good Neighbor Grocery Orange Juice Sales for Year **232**
Table 6.4 Good Neighbor Grocery Orange Juice and Hot Chocolate Sales **241**
Table 6.5 Presentation Graphics Toolbox Routines **261**
Table 8.1 Statements Used to Exit an Error-Handling Routine **278**
Table 8.2 Program Continuation After BASIC Finds an Error-Handling Routine in the Invocation Path **287**
Table 9.1 Compiler Options for Event Trapping **316**
Table 10.1 Valid BASIC Data Types Within ISAM Files **333**
Table 10.2 Executable File for ISAM Programs **379**
Table 11.1 Storage Allocation with Near and Far Strings **403**
Table 11.2 Use of FRE Function for Near and Far Strings **407**
Table 12.1 Language Equivalents for Routine Calls **420**
Table 12.2 Default Methods for Passing Parameters **427**
Table 12.3 Default Naming and Calling Conventions **450**
Table 12.4 Equivalent Numeric Data Types **455**
Table 12.5 Comparison of Array Data Storage in BC and QBX **464**
Table 12.6 Equivalent Array Declarations **467**
Table 12.7 Default Segments and Types for Medium Memory Model **487**
Table 13.1 String-Processing Routines **490**
Table 14.1 Changes to BASIC Statements and Functions for Protected Mode **520**
Table 14.2 Environment Options **528**
Table 15.1 Storage of Data in BASIC **534**
Table 15.2 BASIC Stub Files **540**
Table 16.1 Input to the BC Command **556**
Table 17.1 Library Naming Conventions **565**
Table 18.1 LINK Options Supported with BASIC **588**

Table 18.2 LINK Options not Supported with BASIC **590**
Table 18.3 How LINK Fixes Unresolved References **609**
Table 18.4 Stub Files Included with BASIC **611**
Table 19.1 Requirements for Using BASIC Toolbox Files **625**
Table 20.1 Predefined NMAKE Macros **643**
Table 20.2 Predefined Inference Rules **649**
Table A.1 Special Characters **695**
Table B.1 Numeric Data Types **702**
Table B.2 ISAM Data Types **707**
Table B.3 Types of Numeric Constants **708**
Table B.4 Variable-Type Memory Requirements **717**
Table B.5 Storage of Data in BASIC **719**
Table C.1 Relational Operators and Their Functions **734**
Table C.2 BASIC Logical Operators **736**
Table C.3 Values Returned by Logical Operations **736**

Figures

Figure 1.1 Logic of FOR...NEXT Loop with Positive STEP **17**
Figure 1.2 Logic of FOR...NEXT Loop with Negative STEP **18**
Figure 1.3 Logic of WHILE...WEND Loop **23**
Figure 1.4 Logic of DO WHILE...LOOP **24**
Figure 1.5 Logic of DO UNTIL...LOOP **25**
Figure 1.6 Logic of DO...LOOP WHILE **26**
Figure 1.7 Logic of DO...LOOP UNTIL **27**
Figure 2.1 Parameters and Arguments **44**
Figure 3.1 Text Output on Screen **78**
Figure 3.2 Records in Sequential Files **95**
Figure 3.3 Records in a Random-Access File **103**
Figure 5.1 Coordinates of Selected Pixels in Screen Mode 2 **149**
Figure 5.2 How Angles Are Measured for CIRCLE **159**
Figure 5.3 The Aspect Ratio in Screen Mode 1 **163**
Figure 5.4 WINDOW Contrasted with WINDOW SCREEN **168**
Figure 6.1 Column Chart with Characteristics Labelled **225**
Figure 6.2 Side-by-Side and Stacked Styles for Typical Column Chart **231**
Figure 6.3 Pie Chart **236**
Figure 6.4 Bar Chart **239**
Figure 6.5 Line Chart **240**
Figure 6.6 Column Chart **241**
Figure 6.7 Chart Produced by PGSCAT.BAS **244**
Figure 6.8 Presentation Graphics Toolbox Palettes **255**
Figure 6.9 Multi-Series Chart **264**
Figure 7.1 Main Module Showing Module-Level Code and Procedures **269**
Figure 8.1 Program Flow with RESUME and RESUME NEXT **279**
Figure 10.1 Records and Fields (Rows and Columns) **324**
Figure 10.2 Indexes on the Columns of a Table **325**
Figure 10.3 Insertion and Deletion of Records in a Table **326**
Figure 10.4 Table with Four Rows and Six Columns **330**
Figure 10.5 Presentation Order of Table Assorted by Sex Column **330**
Figure 10.6 Presentation Order of Table Sorted by Sports Column **331**
Figure 10.7 Presentation Order of Table Sorted First by Phone Column, then by Birthday Column **331**
Figure 10.8 The BookStock Table in the BOOKS Database **332**
Figure 10.9 The CardHolders Table **366**
Figure 10.10 The BooksOut Table in BOOKS.MDB **367**
Figure 10.11 Relationship of Tables in LIBRARY.MDB Database **368**
Figure 12.1 Mixed-Language Call **419**
Figure 12.2 Programming with Mixed Languages **422**
Figure 12.3 BASIC/FORTRAN/Pascal Call to Calc (A,B,C) **424**
Figure 12.4 C Call to Calc (A,B,C) **425**

Figure 12.5 C Call to BASIC 446
Figure 12.6 FORTRAN Call to BASIC 449
Figure 12.7 BASIC Near String-Descriptor Format 456
Figure 12.8 C String Format 457
Figure 12.9 FORTRAN String Format 458
Figure 12.10 Pascal Variable-Length String Format 458
Figure 12.11 Column-Major storage 468
Figure 12.12 Row-Major storage 468
Figure 12.13 Structure and Record Storage 470
Figure 12.14 The Stack Frame 480
Figure 12.15 BASIC Return Values 482
Figure 12.16 BASIC Stack Frame 485
Figure 15.1 BASIC Memory Map 533
Figure 17.1 Making a Quick Library from QBX 569

Introduction

This manual deals with advanced programming topics and command-line tools. The focus throughout is on utility, not theory—how you can solve the full range of programming problems with Microsoft BASIC Professional Development System™.

Using This Manual

This manual is divided into two parts. Part 1, "Selected Programming Topics," provides information on specific programming techniques and strategies. Part 2, "Using BASIC Utilities," contains important reference material. Note that the Microsoft BASIC language definition, formerly a part of the *BASIC Language Reference*, is also included in this manual as Appendixes A–D.

Selected Programming Topics

If you have used previous versions of Microsoft BASIC Compiler and QuickBASIC, you'll find that Chapters 1–5 and 7–9 of this manual are those you've known previously as *Programming in BASIC* (Selected Programming Topics). These have now been supplemented with chapters on Presentation Graphics and completely rewritten chapters on error and event handling that show you how to make best use of these BASIC features. Additional advanced programming information is contained in the second half of Part 1, including:

- Database programming with ISAM
- Advanced string storage
- Mixed-language programming
- Mixed-language programming with far strings
- OS/2 programming
- Optimizing program size and speed

These chapters progress from simple to more complex topics. Each chapter contains a variety of short programming examples to explain specific parts of the topic, but most also contain extended examples that demonstrate the chapter's programming principles in the context of complete working programs. For your convenience, you can also find these programs on your distribution disks. (See the file PACKING.LST to locate any materials on the disks.)

Regardless of your interests or background, the chapters in Part 1 will help you learn almost everything you need to know to write sophisticated BASIC applications.

Using BASIC Utilities

The chapters in Part 2 describe how to use the compiler and utilities included in the Microsoft BASIC package. The following table describes each of the chapters.

Chapter	Description
16, "Compiling with BC"	Describes the features, options, and usage of the BASIC Compiler (BC).
17, "About Linking and Libraries"	Provides an overview of linking and libraries for BASIC and mixed-language programs.
18, "Using LINK and LIB"	Describes the Microsoft Segmented-Executable Linker (LINK) and the Microsoft Library Manager (LIB).
19, "Creating and Using Quick Libraries"	Gives instructions for creating Quick libraries and describes how to load and use Quick libraries within the QBX environment.
20, "NMAKE"	Discusses NMAKE, a program maintenance utility.
21, "Building Custom Run-time Modules"	Describes the BASIC Run-Time Module Builder (BUILDRTM).
22, "Customizing Online Help"	Describes the structure of an online Help file and shows how to create or modify Help files using the HELPMAKE utility.

Appendixes

The BASIC language definition is broken down into the following four appendixes:

- Language elements
- Data Types, constants, variables, and arrays
- Operators and expressions
- Modules and procedures

Document Conventions

This manual uses the following typographic conventions.

Example of convention	Description
BASIC.LIB, COPY, LINK, /X	Uppercase (capital) letters indicate filenames and DOS-level commands. Uppercase is also used for command-line options (unless the application accepts only lowercase).
SUB, IF, LOOP, WHILE, TIME$	Bold uppercase letters indicate language-specific keywords with special meaning to Microsoft BASIC or another Microsoft language. Keywords are a required part of statement syntax, unless they are enclosed in double brackets as explained later in this table. In programs you write, you must enter keywords exactly as shown. However, you can use uppercase letters or lowercase letters.
`CALL NewProc(arg1!, var2%)`	This type is used for program examples, program output, and error messages.
	This symbol appears next to examples that are included on the Microsoft BASIC distribution disks.
`Enter a value 1-5: 5`	Bold in an example indicates a response to a prompt.

`'$INCLUDE:'BC.BI'`
`  .`
`  .`
`  .`
`CHAIN "PROG1"`
`END`
	A column of three dots in an example indicates that part of the example program has been intentionally omitted.

`' Make one pass`	The apostrophe (single right quotation mark) marks the beginning of a comment in examples.
filespec$	Italic letters indicate placeholders for information you must supply, such as a filename. The last character of the placeholder often indicates the data type of the information you must supply. Italics are also occasionally used in the text for emphasis.
[[*expressionlist*]]	Items inside double square brackets are optional.

{WHILE | UNTIL}	Braces and a vertical bar indicate a mandatory choice among two or more items. You must choose one of the items unless all of the items are also enclosed in double square brackets.
[[*directory*]]...	Three dots following an item indicate that more items having the same form may appear.
Ctrl key	Initial capital letters are used for the names of keys and key sequences, such as Del and Ctrl+R. The key names used in this manual correspond to the names on the IBM Personal Computer keys. Other machines may use different names.
Alt+F1	A plus (+) indicates a combination of keys. For example, Alt+F1 means to hold down the Alt key while pressing the F1 key. The carriage-return key, sometimes marked with a bent arrow, is referred to as Enter. The cursor movement ("arrow") keys on the numeric keypad are called direction keys. Individual direction keys are referred to by the direction of the arrow on the key top (Left Arrow, Right Arrow, Up Arrow, Down Arrow) or the name on the key top (PgUp, PgDn).
"defined term"	Quotation marks usually indicate a new term defined in the text.
Video Graphics Array (VGA)	Acronyms and abbreviations are usually spelled out the first time they are used.

The following syntax (for the **LOCK...UNLOCK** statement) illustrates many of the typographic conventions in this manual:

LOCK [[#]] *filenumber%* [[,{*record&* | [[*start&*]] **TO** *end&*}]]
.
.
.
UNLOCK [[#]] *filenumber%* [[,{*record&* | [[*start&*]] **TO** *end&*}]]

Note Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft Operating System/2 (MS® OS/2) and IBM OS/2. Similarly, the term "DOS" refers to the MS-DOS and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system. The term "BASICA" refers to interpreted versions of BASIC in general.

Programming Style in This Manual

The following guidelines were used in writing programs in this manual and on the distribution disks. These guidelines are only recommendations for program readability; you are not obliged to follow them when writing your own programs.

- Keywords and symbolic constants appear in uppercase letters:

```
' PRINT, DO, LOOP, UNTIL are keywords:
PRINT "Title Page"
DO LOOP UNTIL Response$ = "N"

' FALSE and TRUE are symbolic constants
' equal to 0 and -1, respectively:
CONST FALSE = 0, TRUE = NOT FALSE
```

- Variable names are lowercase with an initial capital letter; variable names with more than one syllable may contain other capital letters to clarify the division:

```
NumRecords% = 45
DateOfBirth$ = "11/24/54"
```

- Line labels are used instead of line numbers. The use of line labels is restricted to event-trapping and error-handling routines, as well as **DATA** statements when used with **RESTORE**:

```
' TimerHandler and ScreenTwoData are line labels:
ON TIMER GOSUB TimerHandler
RESTORE ScreenTwoData
```

- As noted in the preceding section, a single apostrophe (') introduces comments:

```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Control-flow blocks and statements in procedures or subroutines are indented from the enclosing code:

```
SUB GetInput STATIC
   FOR I% = 1 TO 10
      INPUT X
      IF X > 0 THEN
         .
         .
         .
      ELSE
         .
         .
         .
      END IF
   NEXT I%
END SUB
```

- Lines longer than 80 characters may be continued on the next line using a line-continuation character (_). This is done to fit the example on the printed page. BC and QBX accommodate files containing line-continuation characters. When a continued line is loaded into the QBX environment, QBX removes the underscore and rejoins the line.

Part 1

Selected Programming Topics

Part 1 contains information about using Microsoft BASIC to write programs and create complex applications.

Chapters 1–4 describe elements of the BASIC language. Chapter 1 discusses the control-flow structures that direct your program's execution. Chapter 2 discusses **SUB** and **FUNCTION** procedures, two important building blocks for BASIC programs. Chapter 3 shows you how to use input and output functions and statements to access data stored in files and to communicate with external devices. Chapter 4 describes techniques you can use to manipulate string data.

Chapters 5 and 6 contain information about creating graphics. Chapter 5 shows you how to use Microsoft BASIC graphics statements and functions to create and animate shapes, colors, and patterns on your screen. Chapter 6 shows you how to use the Presentation Graphics toolbox to create pie charts, bar and column charts, line graphs, and scatter diagrams.

Chapter 7 explains how to use modules in your programs. You'll learn why modular programming is useful, as well as how to work with modules in the QuickBASIC Extended (QBX) environment.

Chapters 8 and 9 discusses how to enable, trap, and process errors and events in your program.

Chapter 10 discusses database programming with Indexed Sequential Access Method (ISAM). You'll learn what ISAM is, how to use it to create and maintain databases, and how to search for and retrieve data from an ISAM file.

Chapters 11–13 contain information about using far strings in BASIC programs and about mixed language programming using near and far strings.

Chapter 14 contains information about writing OS/2 protected-mode applications in BASIC.

Chapter 15 provides programming tips for optimizing the speed and size of BASIC programs.

Chapter 1
Control-Flow Structures

This chapter shows you how to use control-flow structures—specifically, loops and decision statements—to control the flow of your program's execution. Loops make a program execute a sequence of statements as many times as you want. Decision statements let the program decide which of several alternative paths to take.

When you are finished with this chapter, you will know how to do the following tasks related to using loops and decision statements in your Microsoft BASIC programs:

- Compare expressions using relational operators.
- Combine string or numeric expressions with logical operators and determine whether the resulting expression is true or false.
- Create branches in the flow of the program with the statements **IF...THEN...ELSE** and **SELECT CASE.**
- Write loops that repeat statements a specific number of times.
- Write loops that repeat statements while or until a certain condition is true.

Changing Statement Execution Order

Left unchecked by control-flow statements, a program's logic flows through statements from left to right, top to bottom. While some very simple programs can be written with only this unidirectional flow, most of the power and utility of any programming language comes from its ability to change statement execution order with decision structures and loops.

With a decision structure, a program can evaluate an expression, then branch to one of several different groups of related statements (statement blocks) depending on the result of the evaluation. With a loop, a program can repeatedly execute statement blocks.

If you are coming to Microsoft BASIC after programming in BASICA, you will appreciate the added versatility of these additional control-flow structures:

- The block **IF...THEN...ELSE** statement
- The **SELECT CASE** statement
- The **DO...LOOP** and **EXIT DO** statements
- The **EXIT FOR** statement, which provides an alternative way to exit **FOR...NEXT** loops

Boolean Expressions

A Boolean expression is any expression that returns the value "true" or "false." BASIC uses Boolean expressions in certain kinds of decision structures and loops. The following **IF...THEN...ELSE** statement contains a Boolean expression, `X < Y` :

```
IF X < Y THEN CALL Procedure1 ELSE CALL Procedure2
```

In the previous example, if the Boolean expression is true (if the value of the variable `X` is in fact less than the value of the variable `Y`), then `Procedure1` is executed; otherwise (if `X` is greater than or equal to `Y`), `Procedure2` is executed.

The preceding example also demonstrates a common use of Boolean expressions: comparing two other expressions (in this case, `X` and `Y`) to determine the relationship between them. These comparisons are made with the relational operators shown in Table 1.1.

Table 1.1 Relational Operators Used in BASIC

Operator	Meaning
=	Equal
< >	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

You can use these relational operators to compare string expressions. In this case "greater than," "less than," and so on refer to alphabetical order. For example, the following expression is true, since the word `deduce` comes alphabetically before the word `deduct`:

```
"deduce" < "deduct"
```

Boolean expressions also frequently use the "logical operators" **AND**, **OR**, **NOT**, **XOR**, **IMP**, and **EQV**. These operators allow you to construct compound tests from one or more Boolean expressions. For example:

```
expression1 AND expression2
```

The preceding example is true only if *`expression1`* and *`expression2`* are both true. Thus, in the following example, the message `All sorted` is printed only if both the Boolean expressions `X <= Y` and `Y <= Z` are true:

```
IF (X <= Y) AND (Y <= Z) THEN PRINT "All sorted"
```

The parentheses around the Boolean expressions in the last example are not really necessary, since relational operators such as **<=** are evaluated before logical operators such as **AND**.

However, parentheses make a complex Boolean expression more readable and ensure that its components are evaluated in the order that you intend.

BASIC uses the numeric values –1 and 0 to represent true and false, respectively. You can see this by asking BASIC to print a true expression and a false expression, as in the next example:

```
X = 5
Y = 10
PRINT X < Y   ' Evaluate, print a "true"  Boolean expression.
PRINT X > Y   ' Evaluate, print a "false" Boolean expression.
```

Output

```
-1
 0
```

The value –1 for true makes more sense when you consider how BASIC's **NOT** operator works: **NOT** inverts each bit in the binary representation of its operand, changing one bits to zero bits, and zero bits to one bits. Therefore, since the integer value 0 (false) is stored internally as a sequence of 16 zero bits, `NOT 0` (true) is stored internally as 16 one bits, as follows:

```
0000000000000000

TRUE = NOT FALSE = 1111111111111111
```

In the two's-complement method that BASIC uses to store integers, 16 one bits represent the value –1.

Note that BASIC returns –1 when it evaluates a Boolean expression as true; however, BASIC considers any nonzero value to be true, as shown by the output from the following example:

```
INPUT "Enter a value: ", X

IF X THEN PRINT X;"is true."
```

Output

```
Enter a value: 2
 2 is true.
```

The **NOT** operator in BASIC is a "bitwise" operator. Some programming languages, such as C and Pascal, have both a bitwise **NOT** operator and a "logical" **NOT** operator. The distinction is as follows:

- A bitwise **NOT** returns false (0) only for the value –1.
- A logical **NOT** returns false (0) for any true (nonzero) value.

In BASIC, for any true *expression* not equal to –1, **NOT** *expression* returns another true value, as shown in the following table:

Value of expression	Value of NOT expression
1	–2
2	–3
–2	1
–1	0

So beware: **NOT** *expression* is false only if *expression* evaluates to a value of –1. If you define Boolean constants or variables for use in your programs, use –1 for true.

You can use the values 0 and –1 to define helpful mnemonic Boolean constants for use in loops or decisions. This technique is used in many of the examples in this manual, as shown in the following program fragment, which sorts the elements of an array named `Amount` in ascending order:

```
' Define symbolic constants to use in program:
CONST FALSE = 0, TRUE = NOT FALSE
.
.
.
DO
    Swaps% = FALSE
    FOR I% = 1 TO TransacNum - 1
        IF Amount(I%) < Amount(I%+1) THEN
            SWAP Amount(I%), Amount(I%+1)
            Swaps% = TRUE
        END IF
    NEXT I%
LOOP WHILE Swaps%    ' Keep looping while Swaps is TRUE.
.
.
.
```

Decision Structures

Based on the value of an expression, decision structures cause a program to take one of the following two actions:

- Execute one of several alternative statements within the decision structure itself.
- Branch to another part of the program outside the decision structure.

In BASICA, decision-making is handled solely by the single-line **IF...THEN**[**...ELSE**] statement. In its simplest form (**IF...THEN**), the expression following the **IF** keyword is evaluated. If the expression is true, the program executes the statements following the **THEN** keyword; if the expression is false, the program continues with the next line after the

IF...THEN statement. Lines 50 and 70 from the following BASICA program fragment show examples of **IF...THEN**:

```
30  INPUT A
40  ' If A is greater than 100, print a message and branch
45  ' back to line 30; otherwise, go on to line 60:
50  IF A > 100 THEN PRINT "Too big": GOTO 30
60  ' If A is equal to 100, branch to line 300;
65  ' otherwise, go on to line 80:
70  IF A = 100 THEN GOTO 300
80  PRINT A/100: GOTO 30
.
.
.
```

By adding the **ELSE** clause to an **IF...THEN** statement, you can have your program take one set of actions (those following the **THEN** keyword) if an expression is true, and another set of actions (those following the **ELSE** keyword) if it is false. The next program fragment shows how **ELSE** works in an **IF...THEN...ELSE** statement:

```
10 INPUT "What is your password"; Pass$
15 ' If user enters "sword," branch to line 50;
20 ' otherwise, print a message and branch back to line 10:
30 IF Pass$="sword" THEN 50 ELSE PRINT "Try again": GOTO 10
```

While BASICA's single-line **IF...THEN...ELSE** is adequate for simple decisions, it can lead to virtually unreadable code in cases of more complicated decisions. This is especially true if you write your programs so all alternative actions take place within the **IF...THEN...ELSE** statement itself or if you nest **IF...THEN...ELSE** statements (that is, if you put one **IF...THEN...ELSE** inside another, a perfectly legal construction). As an example of how difficult it is to follow even a simple test, consider the next fragment from a BASICA program:

```
10 ' The following nested IF...THEN...ELSE statements print
15 ' different output for each of the following four cases:
20 '  1) A <= 50, B <= 50     3)  A > 50, B <= 50
25 '  2) A <= 50, B > 50      4)  A > 50, B > 50
30
35 INPUT A, B
40
45 IF A <= 50 THEN IF B <= 50 THEN PRINT "A <= 50, B <= 50" ELSE PRINT
"A <= 50, B > 50" ELSE IF B <= 50 THEN PRINT "A > 50, B <= 50;" ELSE
PRINT "A > 50, B > 50"
```

Even though line 45 extends over several physical lines on the screen, it is just one logical line (everything typed before the Enter key was pressed). BASICA wraps long lines on the screen. To avoid the kind of complicated statement shown by the preceding example, BASIC now includes the block form of the **IF...THEN...ELSE** statement, so that a decision is no longer

restricted to one logical line. The following example shows the same BASICA program rewritten to use block IF...THEN...ELSE:

```
INPUT A, B
IF A <= 50 THEN
   IF B <= 50 THEN
      PRINT "A <= 50, B <= 50"
   ELSE
      PRINT "A <= 50, B > 50"
   END IF
ELSE
   IF B <= 50 THEN
      PRINT "A > 50, B <= 50"
   ELSE
      PRINT "A > 50, B > 50"
   END IF
END IF
```

Microsoft BASIC also provides the **SELECT CASE...END SELECT** (referred to as **SELECT CASE**) statement for structured decisions.

The block **IF...THEN...ELSE** statement and the **SELECT CASE** statement allow the appearance of your code to be based on program logic, rather than requiring many statements to be crowded onto one line. This gives you increased flexibility while you are programming, as well as improved program readability and ease of maintenance when you are done.

Block IF...THEN...ELSE

Table 1.2 shows the syntax of the block **IF...THEN...ELSE** statement and gives an example of its use.

Table 1.2 Block IF...THEN...ELSE Syntax and Example

Syntax	Example

IF *condition1* **THEN**
  [[*statementblock-1*]]
	`IF X > 0 THEN`
	`  PRINT "X is positive"`
	`  PosNum = PosNum + 1`

[[**ELSEIF** *condition2* **THEN**
  [[*statementblock-2*]]]]
	`ELSEIF X < 0 THEN`
	`  PRINT "X is negative"`
	`  NegNum = NegNum + 1`

[[**ELSE** [[*statementblock-n*]]]]	`ELSE` `  PRINT "X is zero"`
END IF	`END IF`

The arguments *condition1*, *condition2*, and so on are expressions. They can be any numeric expression (in which case true becomes any nonzero value, and false is 0), or they can be Boolean expressions (in which case true is –1 and false is 0). As explained previously, Boolean expressions typically compare two numeric or string expressions using one of the relational operators, such as < or >=.

Each **IF**, **ELSEIF**, and **ELSE** clause is followed by a block of statements. None of the statements in the block can be on the same line as the **IF**, **ELSEIF**, or **ELSE** clause; otherwise, BASIC considers it a single-line **IF...THEN** statement.

BASIC evaluates each of the expressions in the **IF** and **ELSEIF** clauses from top to bottom, skipping over statement blocks until it finds the first true expression. When it finds a true expression, it executes the statements corresponding to the expression, then branches out of the block to the statement following the **END IF** clause.

If none of the expressions in the **IF** or **ELSEIF** clauses is true, BASIC skips to the **ELSE** clause, if there is one, and executes its statements. Otherwise, if there is no **ELSE** clause, the program continues with the next statement after the **END IF** clause.

The **ELSE** and **ELSEIF** clauses are optional, as shown in the following example:

```
' If the value of X is less than 100, execute the two statements
' before END IF; otherwise, go to the INPUT statement
' following END IF:

IF X < 100 THEN
   PRINT X
   Number = Number + 1
END IF
INPUT "New value"; Response$
.
.
.
```

A single block **IF...THEN...ELSE** can contain multiple **ELSEIF** statements, as follows:

```
IF C$ >= "A" AND C$ <= "Z" THEN
   PRINT "Capital letter"
ELSEIF C$ >= "a" AND C$ <= "z" THEN
   PRINT "Lowercase letter"
ELSEIF C$ >= "0" AND C$ <= "9" THEN
   PRINT "Number"
ELSE
   PRINT "Not alphanumeric"
END IF
```

At most, only one block of statements is executed, even if more than one condition is true. For example, if you enter the word **ace** as input to the next example, it prints the message `Input too short` but does not print the message `Can't start with an a`.

```
INPUT Check$
IF LEN(Check$) > 6 THEN
   PRINT "Input too long"
ELSEIF LEN(Check$) < 6 THEN
   PRINT "Input too short"
ELSEIF LEFT$(Check$, 1) = "a" THEN
   PRINT "Can't start with an a"
END IF
```

IF...THEN...ELSE statements can be nested; in other words, you can put an **IF...THEN...ELSE** statement inside another **IF...THEN...ELSE** statement, as shown here:

```
IF X > 0 THEN
   IF Y > 0 THEN
      IF Z > 0 THEN
   PRINT "All are greater than zero."
      ELSE
   PRINT "Only X and Y greater than zero."
      END IF
   END IF
ELSEIF X = 0 THEN
   IF Y = 0 THEN
      IF Z = 0 THEN
   PRINT "All equal zero."
      ELSE
   PRINT "Only X and Y equal zero."
      END IF
   END IF
ELSE
   PRINT "X is less than zero."
END IF
```

SELECT CASE

The **SELECT CASE** statement is a multiple-choice decision structure similar to the block **IF...THEN...ELSE** statement. **SELECT CASE** can be used anywhere block **IF...THEN...ELSE** can be used.

The difference between the two is that **SELECT CASE** evaluates a single expression, then executes different statements or branches to different parts of the program based on the result. In contrast, a block **IF...THEN...ELSE** can evaluate completely different expressions.

Examples

The following examples illustrate the similarities and differences between the **SELECT CASE** and **IF...THEN...ELSE** statements. Here is an example of using block **IF...THEN...ELSE** for a multiple-choice decision:

```
INPUT X
IF X = 1 THEN
   PRINT "One"
ELSEIF X = 2 THEN
   PRINT "Two"
ELSEIF X = 3 THEN
   PRINT "Three"
ELSE
   PRINT "Must be integer from 1 to 3."
END IF
```

The previous decision is rewritten using **SELECT CASE** as follows:

```
INPUT X
SELECT CASE X
   CASE 1
      PRINT "One"
   CASE 2
      PRINT "Two"
   CASE 3
      PRINT "Three"
   CASE ELSE
      PRINT "Must be integer from 1 to 3."
END SELECT
```

The following decision can be made either with the **SELECT CASE** or the block **IF...THEN...ELSE** statement. The comparison is more efficient with the **IF...THEN...ELSE** statement because different expressions are being evaluated in the **IF** and **ELSEIF** clauses.

```
INPUT X, Y
IF X = 0 AND Y = 0 THEN
   PRINT "Both are zero."
ELSEIF X = 0 THEN
   PRINT "Only X is zero."
ELSEIF Y = 0 THEN
   PRINT "Only Y is zero."
ELSE
   PRINT "Neither is zero."
END IF
```

Using the SELECT CASE Statement

Table 1.3 shows the syntax of a **SELECT CASE** statement and an example.

Table 1.3 SELECT CASE Syntax and Example

Syntax	Example

SELECT CASE *testexpression*
  CASE *expressionlist1*
  [[*statementblock-1*]]
  [[**CASE** *expressionlist2*
  [[*statementblock-2*]]]]
  .
  .
  [[**CASE ELSE**
  [[*statementblock-n*]]]]
END SELECT
	`INPUT TestValue`
	`SELECT CASE TestValue`
	`  CASE 1, 3, 5, 7, 9`
	`  PRINT "Odd"`
	`  CASE 2, 4, 6, 8`
	`  PRINT "Even"`
	`  CASE IS < 1`
	`  PRINT "Too low"`
	`  CASE IS > 9`
	`  PRINT "Too high"`
	`  CASE ELSE`
	`  PRINT "Not an integer"`
	`END SELECT`

The *expressionlist* arguments following a **CASE** clause can be one or more of the following, separated by commas:

- A numeric expression or a range of numeric expressions
- A string expression or a range of string expressions

To specify a range of expressions, use the following syntax for the **CASE** statement:

CASE *expression* **TO** *expression*
CASE IS *relational-operator expression*

The *relational-operator* is any of the operators shown in Table 1.1. For example, if you use `CASE 1 TO 4`, the statements associated with this case are executed when the *testexpression* in the **SELECT CASE** statement is greater than or equal to 1 and less than or equal to 4. If you use `CASE IS < 5`, the associated statements are executed only if *testexpression* is less than 5.

If you are expressing a range with the **TO** keyword, be sure to put the lesser value first. For example, if you want to test for a value from –5 to –1, write the **CASE** statement as follows:

```
CASE -5 TO -1
```

However, the following statement is not a valid way to specify the range from –5 to –1, so the statements associated with this case are never executed:

```
CASE -1 TO -5
```

Similarly, a range of string constants should list the string that comes first alphabetically:

```
CASE "aardvark" TO "bear"
```

Multiple expressions or ranges of expressions can be listed for each **CASE** clause, as in the following lines:

```
CASE 1 TO 4, 7 TO 9, WildCard1%, WildCard2%
CASE IS = Test$, IS = "end of data"
CASE IS < LowerBound, 5, 6, 12, IS > UpperBound
CASE IS < "HAN", "MAO" TO "TAO"
```

If the value of the **SELECT CASE** expression appears in the list following a **CASE** clause, only the statements associated with that **CASE** clause are executed. Control then jumps to the first executable statement following **END SELECT**, not the next block of statements inside the **SELECT CASE** structure, as shown by the output from the next example:

```
INPUT X
SELECT CASE X
   CASE 1
      PRINT "One, ";
   CASE 2
      PRINT "Two, ";
   CASE 3
      PRINT "Three, ";
END SELECT
PRINT "that's all"
```

Output

```
? 1
One, that's all
```

If the same value or range of values appears in more than one **CASE** clause, only the statements associated with the first occurrence are executed, as shown by the next example's output:

```
INPUT Test$
SELECT CASE Test$
   CASE "A" TO "AZZZZZZZZZZZZZZZZZ"
      PRINT "An uppercase word beginning with A"
   CASE IS < "A"
      PRINT "Some sequence of nonalphabetic characters"
```

```
    CASE "ABORIGINE"
       ' This case is never executed since ABORIGINE
       ' falls within the range in the first CASE clause:
       PRINT "A special case"
END SELECT
```

Output

```
? ABORIGINE
An uppercase word beginning with A
```

If you use a **CASE ELSE** clause, it must be the last **CASE** clause listed in the **SELECT CASE** statement. The statements between a **CASE ELSE** clause and an **END SELECT** clause are executed only if the *testexpression* argument does not match any of the other **CASE** selections in the **SELECT CASE** statement. In fact, it is a good idea to have a **CASE ELSE** statement in your **SELECT CASE** block to handle unforeseen values for *testexpression*. However, if there is no **CASE ELSE** statement and no match is found in any **CASE** statement for *testexpression*, the program continues execution.

Example

The following program fragment demonstrates a common use of the **SELECT CASE** statement. It prints a menu on the screen, then branches to different subprograms based on the number typed by the user.

```
DO              ' Start menu loop.

   CLS          ' Clear screen.

   ' Print five menu choices on the screen:
   PRINT "MAIN MENU" : PRINT
   PRINT "1)  Add New Names"
   PRINT "2)  Delete Names"
   PRINT "3)  Change Information"
   PRINT "4)  List Names"
   PRINT
   PRINT "5)  EXIT"

   ' Print input prompt:
   PRINT : PRINT "Type your selection (1 to 5):"

   ' Wait for the user to press a key. INPUT$(1)
   ' reads one character input from the keyboard:
   Ch$ = INPUT$(1)

   ' Use SELECT CASE to process the response:
   SELECT CASE Ch$
```

```
        CASE "1"
            CALL AddData
        CASE "2"
            CALL DeleteData
        CASE "3"
            CALL ChangeData
        CASE "4"
            CALL ListData
        CASE "5"
            EXIT DO' The only way to exit the menu loop.
        CASE ELSE
            BEEP
    END SELECT

LOOP            ' End menu loop.

END

' Subprograms AddData, DeleteData, ChangeData, and ListData:
.
.
.
```

SELECT CASE Vs. ON...GOSUB

You can use the more versatile **SELECT CASE** statement as a replacement for the old **ON...GOSUB** statement. The **SELECT CASE** statement has many advantages over the **ON...GOSUB** statement, summarized as follows:

- The *testexpression* in **SELECT CASE** can evaluate to either a string or numeric value. The *expression* given in the statement **ON...GOSUB** must evaluate to a number within the range 0 to 255.
- The **SELECT CASE** statement branches to a statement block immediately following the matching **CASE** clause. In contrast, **ON...GOSUB** branches to a subroutine in another part of the program.
- **CASE** clauses can be used to test *expression* against a range of values. When the range is quite large, this is not easily done with **ON...GOSUB**, especially in cases such as those shown in the code fragments in the rest of this section.

In the following fragment, the **ON...GOSUB** statement branches to one of the subroutines 50, 100, or 150, depending on the value entered by the user:

```
X% = -1
WHILE X%
    INPUT "Enter choice (0 to quit): ", X%
    IF X% = 0 THEN END
```

```
    WHILE X% < 1 OR X% > 12
       PRINT "Must be value from 1 to 12"
       INPUT "Enter choice (0 to quit): ", X%
    WEND
    ON X% GOSUB 50,50,50,50,50,50,50,50,100,100,100,150
WEND
.
.
.
```

Contrast the preceding example with the next one, which uses a **SELECT CASE** statement with ranges of values in each **CASE** clause:

```
DO
    INPUT "Enter choice (0 to quit): ", X%
    SELECT CASE X%
       CASE 0
          END
       CASE 1 TO 8    ' Replaces "subroutine 50"
                      ' in preceding example
       CASE 9 TO 11   ' Replaces "subroutine 100"
                      ' in preceding example
       CASE 12        ' Replaces "subroutine 150"
                      ' in preceding example
       CASE ELSE      ' Input was out of range.
          PRINT "Must be value from 1 to 12"
       END SELECT
LOOP
```

Looping Structures

Looping structures repeat a block of statements (the loop), either for a specified number of times or until a certain expression (the loop condition) is true or false.

Users of BASICA are familiar with two looping structures, **FOR...NEXT** and **WHILE...WEND**, which are discussed in the following sections "FOR...NEXT Loops" and "WHILE...WEND Loops." Microsoft BASIC has extended the available loop structures with **DO...LOOP**.

FOR...NEXT Loops

A **FOR...NEXT** loop repeats the statements enclosed in the loop a specified number of times, counting from a starting value to an ending value by increasing or decreasing a loop counter. As long as the loop counter has not reached the ending value, the loop continues to execute. Table 1.4 shows the syntax of the **FOR...NEXT** statement and gives an example of its use.

Table 1.4 FOR...NEXT Syntax and Example

Syntax	Example
FOR *counter* = *start* **TO** *end* [[**STEP** *increment*]]	`FOR I% = 1 TO 10`
[[*statementblock-1*]]	`  Array%(I%) = I%`
[[**EXIT FOR**	`NEXT`
[[*statementblock-2*]]]]	
NEXT [[*counter*]]	

In a **FOR...NEXT** loop, the *counter* variable initially has the value of the expression *start*. After each repetition of the loop, the value of *counter* is adjusted. If you leave off the optional **STEP** keyword, the default value for this adjustment is 1; that is, 1 is added to *counter* each time the loop executes. If you use **STEP**, then *counter* is adjusted by the amount *increment*. The *increment* argument can be any numeric value; if it is negative, the loop counts down from *start* to *end*. After the *counter* variable is increased or decreased, its value is compared with *end*. At this point, if either of the following is true, the loop is completed:

- The loop is counting up (*increment* is positive) and *counter* is greater than *end*.
- The loop is counting down (*increment* is negative) and *counter* is less than *end*.

Figure 1.1 shows the logic of a **FOR...NEXT** loop when the value of *increment* is positive.

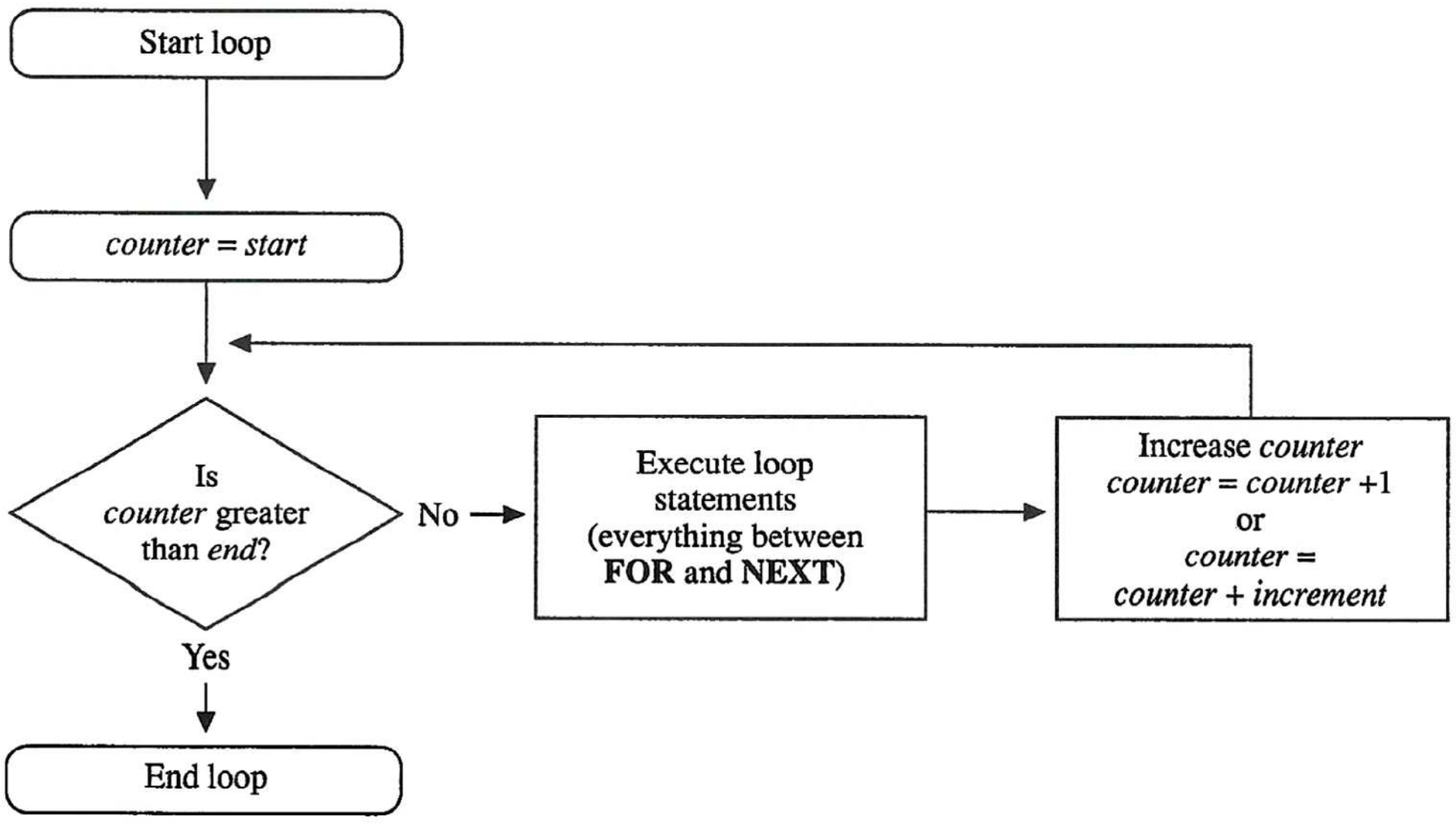


Figure 1.1 Logic of FOR...NEXT Loop with Positive STEP

Figure 1.2 shows the logic of a **FOR...NEXT** loop when the value of *increment* is negative.

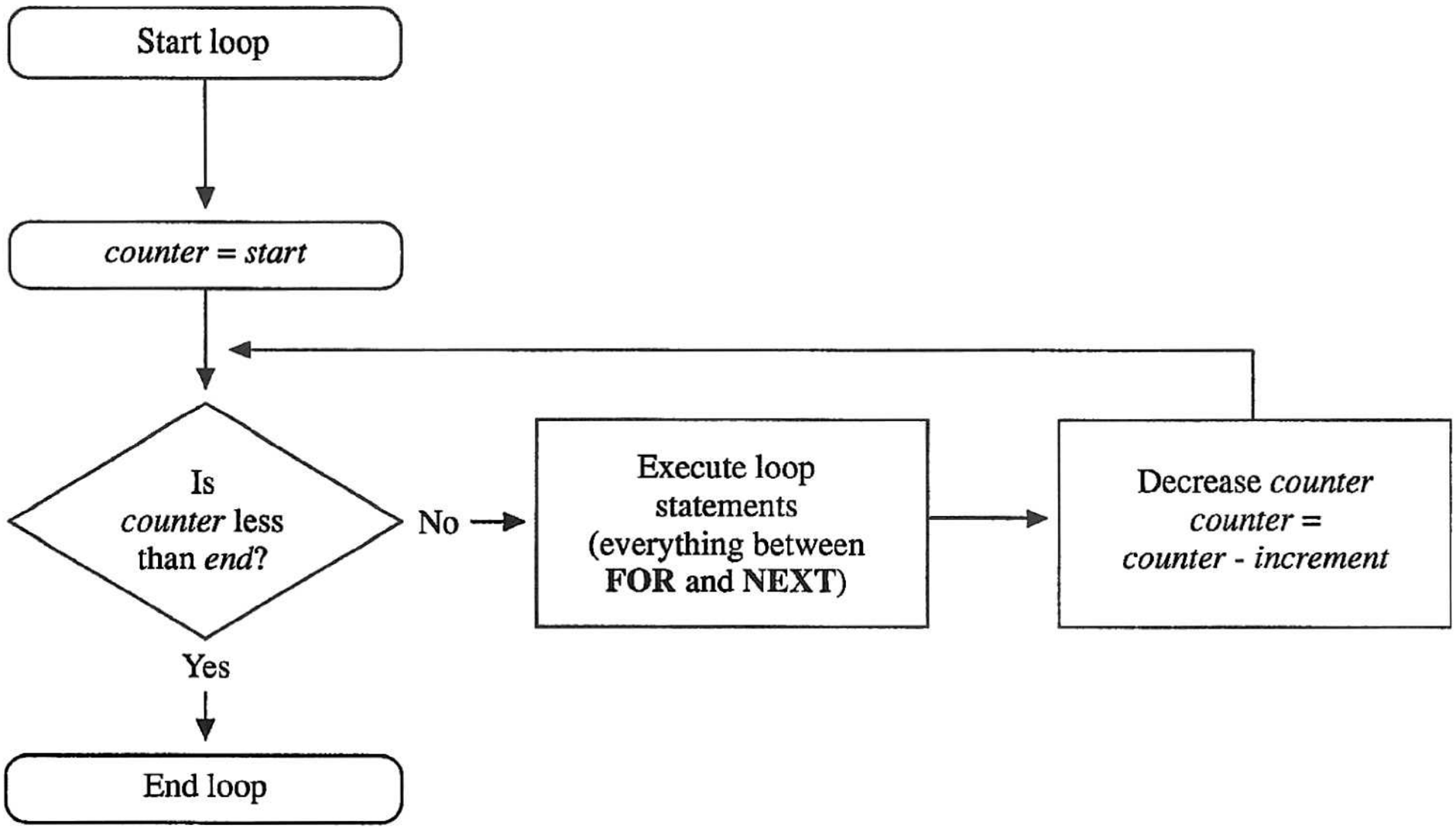


Figure 1.2 Logic of FOR...NEXT Loop with Negative STEP

A **FOR...NEXT** statement always "tests at the top," so if one of the following conditions is true, the loop is never executed:

- The *increment* is positive, and the initial value of *start* is greater than the value of *end*:

```
' Loop never executes, because I% starts out greater
' than 9:
FOR I% = 10 TO 9
   .
   .
   .
NEXT I%
```

- The *increment* is negative, and the initial value of *start* is less than the value of *end*:

```
' Loop never executes, because I% starts out less than 9:
FOR I% =  -10 TO -9 STEP -1
   .
   .
   .
NEXT I%
```

You don't have to use the *counter* argument in the **NEXT** clause; however, if you have several nested **FOR...NEXT** loops (one loop inside another), listing the *counter* arguments can be a helpful way to keep track of what loop you are in.

Here are some general guidelines for nesting **FOR...NEXT** loops:

- If you use a loop counter variable in a **NEXT** clause, the counter for a nested loop must appear before the counter for any enclosing loop. In other words, the following is a legal nesting:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT J
NEXT I
```

  However, the following is not a legal nesting:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT I
NEXT J
```

- For faster loops that generate smaller code, use integer variables for counters in the loops whenever possible.
- If you use a separate **NEXT** clause to end each loop, then the number of **NEXT** clauses must always be the same as the number of **FOR** clauses.
- If you use a single **NEXT** clause to terminate several levels of **FOR...NEXT** loops, then the loop-counter variables must appear after the **NEXT** clause in "inside-out" order:

  NEXT *innermost-loopcounter, ... , outermost-loopcounter*

  In this case, the number of loop-counter variables in the **NEXT** clause must be the same as the number of **FOR** clauses.

Examples The following three program fragments illustrate different ways of nesting **FOR...NEXT** loops to produce the identical output. The first example shows nested **FOR...NEXT** loops with loop counters and separate **NEXT** clauses for each loop:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT K
   NEXT J
NEXT I
```

The following example also uses loop counters but only one **NEXT** clause for all three loops:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
NEXT K, J, I
```

The final example shows nested **FOR...NEXT** loops without loop counters:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT
   NEXT
NEXT
```

Output

```
 1            4            7
 1            4            8
 1            5            7
 1            5            8
 2            4            7
 2            4            8
 2            5            7
 2            5            8
```

Exiting a FOR...NEXT Loop with EXIT FOR

Sometimes you may want to exit a **FOR...NEXT** loop before the counter variable reaches the ending value of the loop. You can do this with the **EXIT FOR** statement. A single **FOR...NEXT** loop can have any number of **EXIT FOR** statements, and the **EXIT FOR** statements can appear anywhere within the loop. The following fragment shows one use for an **EXIT FOR** statement:

```
' Print the square roots of the numbers from 1 to 30,000.
' If the user presses any key while this loop is executing,
' control exits from the loop:
FOR I% = 1 TO 30000
   PRINT SQR(I%)
   IF INKEY$ <> "" THEN EXIT FOR
NEXT
.
.
.
```

EXIT FOR exits only the smallest enclosing **FOR...NEXT** loop in which it appears. For example, if the user presses a key while the following nested loops are executing, the program would simply exit the innermost loop. If the outermost loop is still active (that is, if the value of `I%` is less than or equal to 100), control passes right back to the innermost loop:

```
FOR I% = 1 TO 100
   FOR J% = 1 TO 100
      PRINT I% / J%
      IF INKEY$ <> "" THEN EXIT FOR
   NEXT J%
NEXT I%
```

Suspending Program Execution with FOR...NEXT

Many BASICA programs use an empty **FOR...NEXT** loop such as the following to insert a pause in a program:

```
.
.
.
' There are no statements in the body of this loop;
' all it does is count from 1 to 10,000
' using integers (whole numbers).
FOR I% = 1 TO 10000: NEXT
.
.
.
```

For very short pauses or pauses that do not have to be of some exact interval, using **FOR...NEXT** is fine. The problem with using an empty **FOR...NEXT** loop in this way is that different computers, different versions of BASIC, or different compile-time options can all affect how quickly the arithmetic in a **FOR...NEXT** loop is performed. So the length of a pause can vary widely. BASIC's **SLEEP** statement now provides a better alternative.

WHILE...WEND Loops

The **FOR...NEXT** statement is useful when you know ahead of time exactly how many times a loop should be executed. When you cannot predict the precise number of times a loop should be executed, but do know the condition that will end the loop, the **WHILE...WEND** statement is useful. Instead of counting to determine if it should keep executing a loop, **WHILE...WEND** repeats the loop as long as the loop condition is true.

Table 1.5 shows the syntax of the **WHILE...WEND** statement and an example.

Table 1.5 WHILE...WEND Syntax and Example

Syntax	Example

WHILE *condition*
  [[*statementblock*]]
WEND
	`INPUT R$`
	`WHILE R$ <> "Y" AND R$ <> "N"`
	`  PRINT "Answer must be Y or N."`
	`  INPUT R$`
	`WEND`

Example The following example assigns an initial value of ten to the variable `X`, then successively halves that value and keeps halving it until the value of `X` is less than .00001:

```
X = 10

WHILE X > .00001
   PRINT X
   X = X/2
WEND
```

Figure 1.3 illustrates the logic of a **WHILE...WEND** loop.

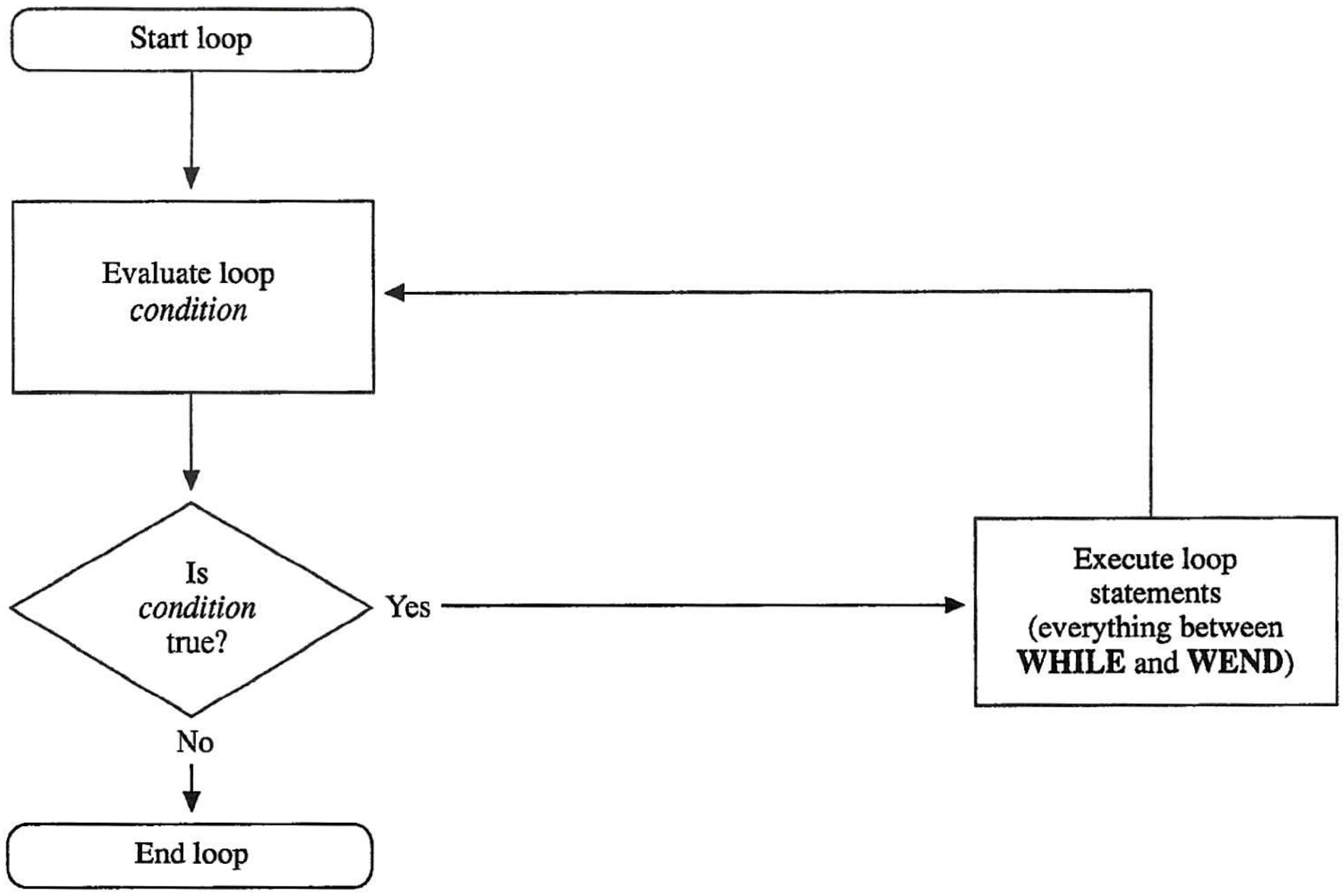


Figure 1.3 Logic of WHILE...WEND Loop

DO...LOOP Loops

Like the **WHILE...WEND** statement, the **DO...LOOP** statement executes a block of statements an indeterminate number of times; that is, exiting from the loop depends on the truth or falsehood of the loop condition. Unlike **WHILE...WEND**, **DO...LOOP** allows you to test for either a true or false condition. You can also put the test at either the beginning or the end of the loop.

Table 1.6 shows the syntax of a loop that tests at the loop's beginning.

Table 1.6 DO...LOOP Syntax and Example: Test at the Beginning

Syntax	Example
DO [{**WHILE** | **UNTIL**} *condition*]	`DO UNTIL INKEY$ <> ""`
[*statementblock-1*]	`' An empty loop that`
[**EXIT DO**	`' pauses until a key`
[*statementblock-2*]]	`' is pressed`
LOOP	`LOOP`

Figures 1.4 and 1.5 illustrate the two kinds of **DO...LOOP** statements that test at the beginning of the loop.

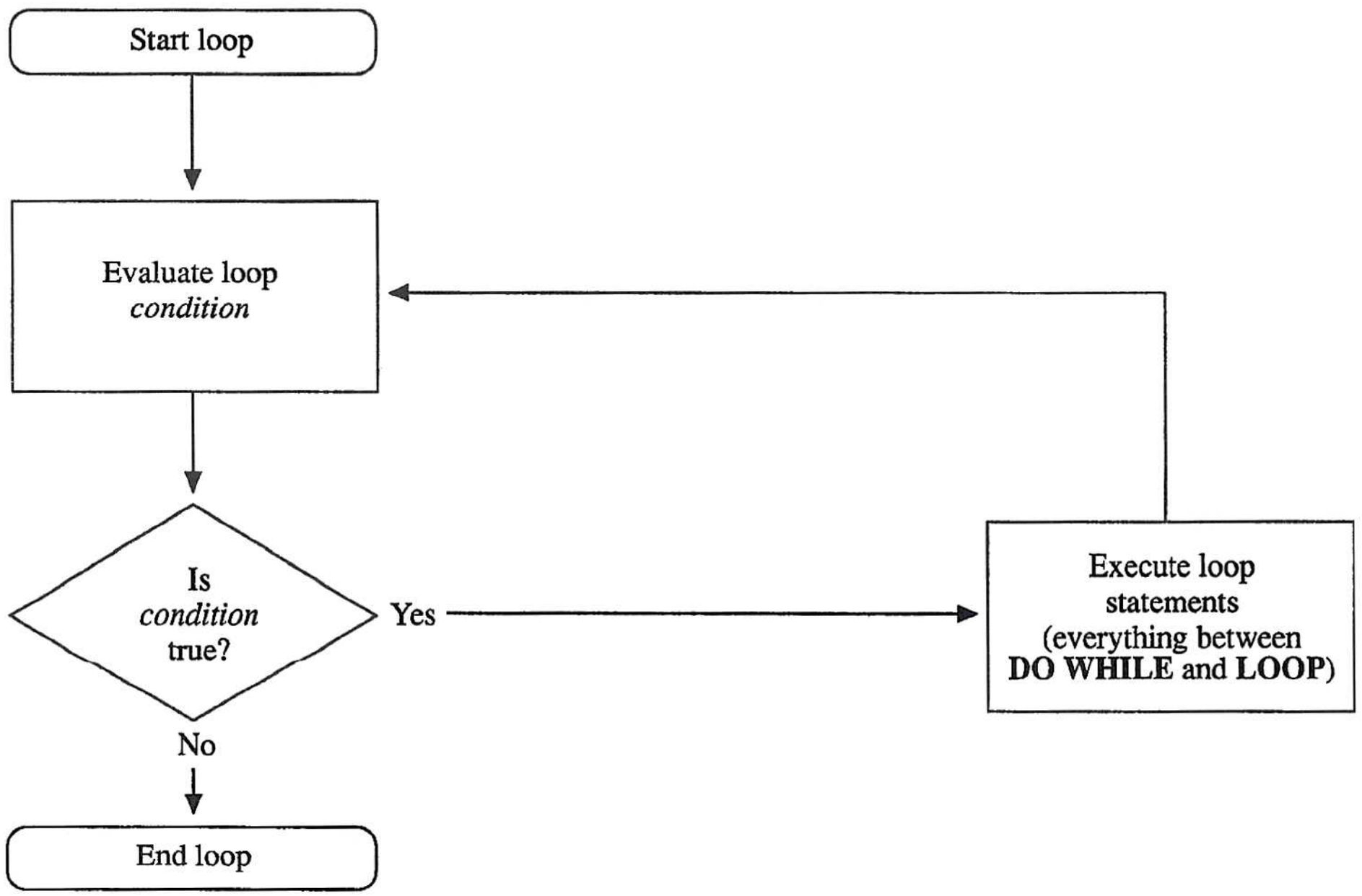


Figure 1.4 Logic of DO WHILE...LOOP

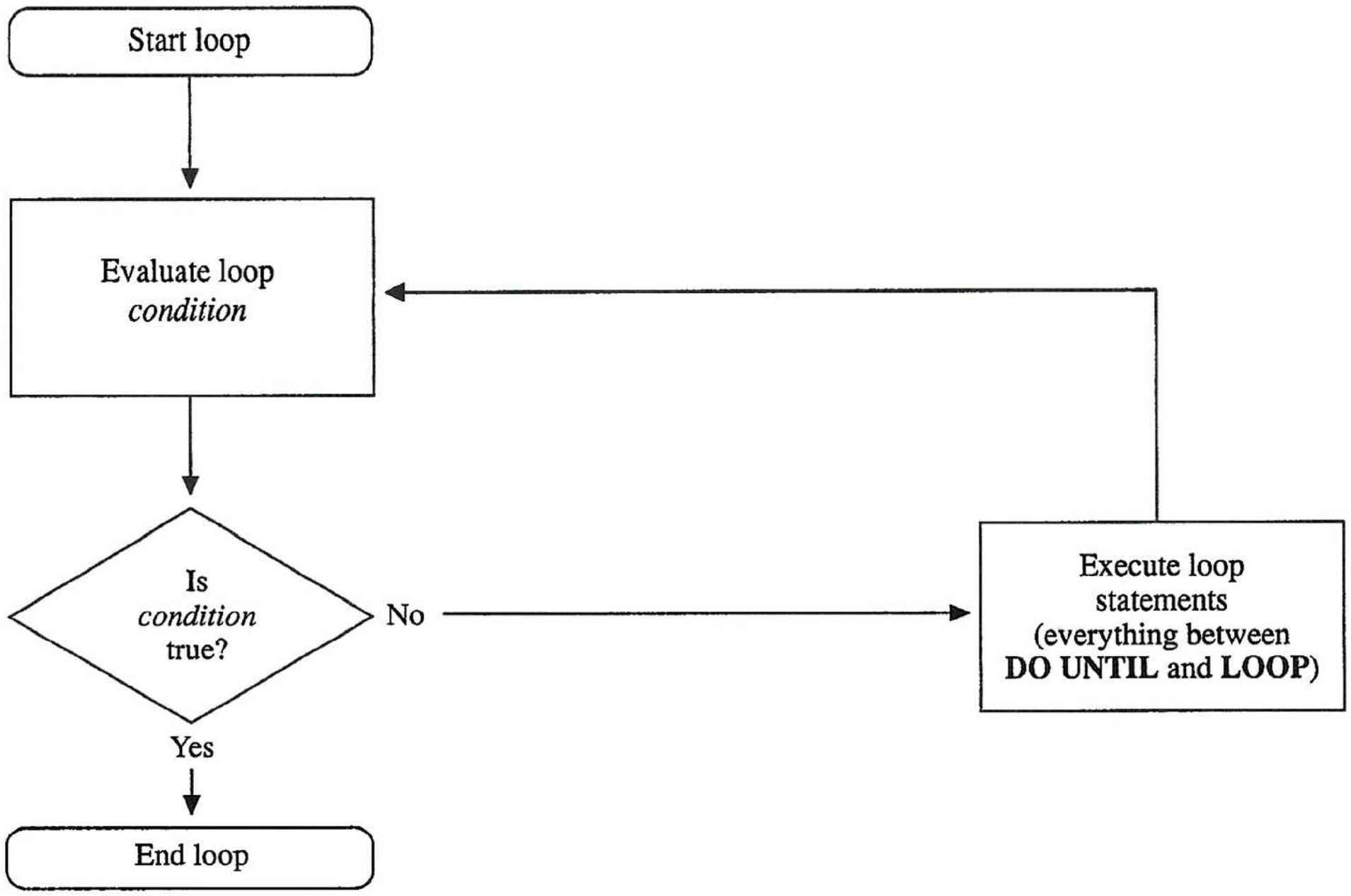


Figure 1.5 Logic of DO UNTIL...LOOP

Table 1.7 shows the syntax of a loop that tests for true or false at the end of the loop.

Table 1.7 DO...LOOP Syntax and Example: Test at the End

Syntax	Example

DO
  [[*statementblock-1*]]
  [[**EXIT DO**
  statementblock-2]]]]
LOOP [[{**WHILE** | **UNTIL**} *condition*]]
	`DO`
	`  INPUT "Change: ", Ch`
	`  Total = Total + Ch`
	`LOOP WHILE Ch <> 0`

Figures 1.6 and 1.7 illustrate the two kinds of **DO...LOOP** statements that test at the end of the loop.

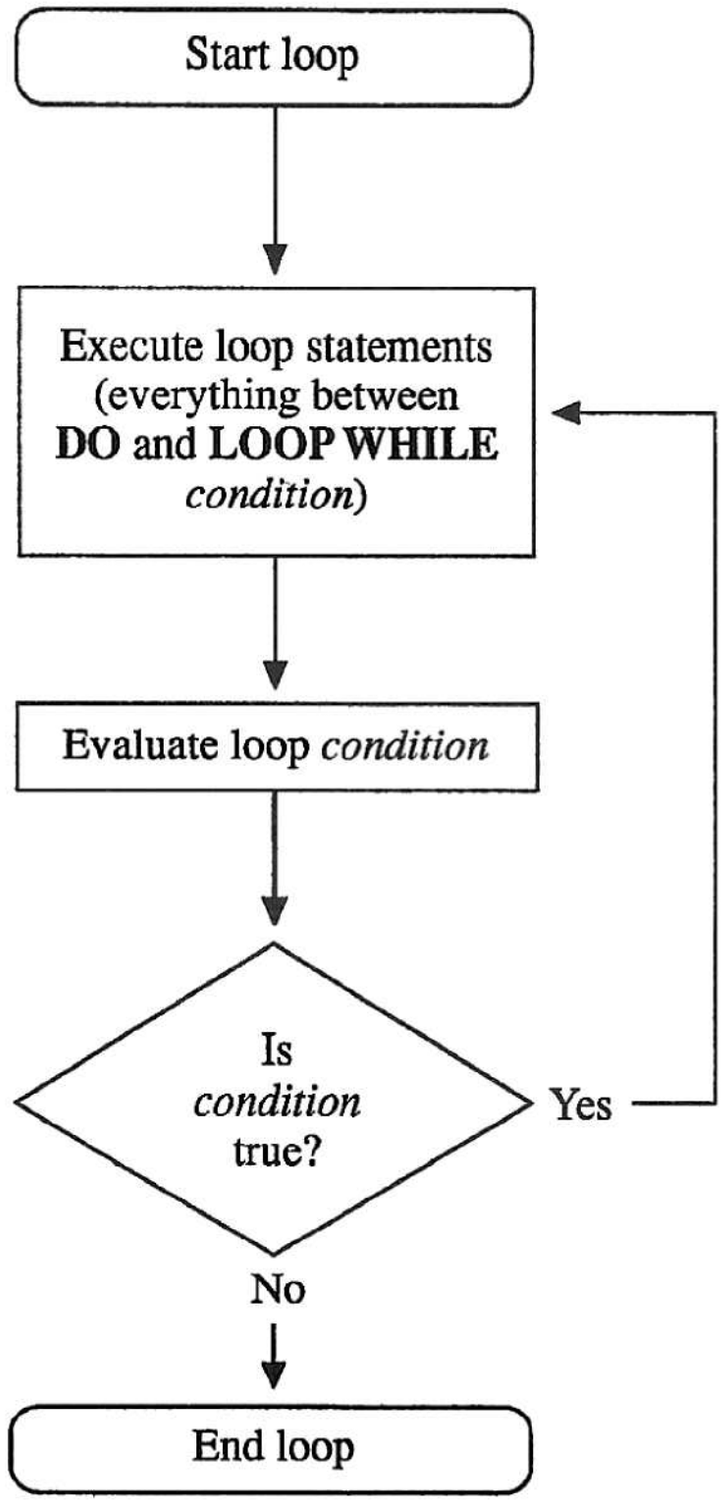


Figure 1.6 Logic of DO...LOOP WHILE

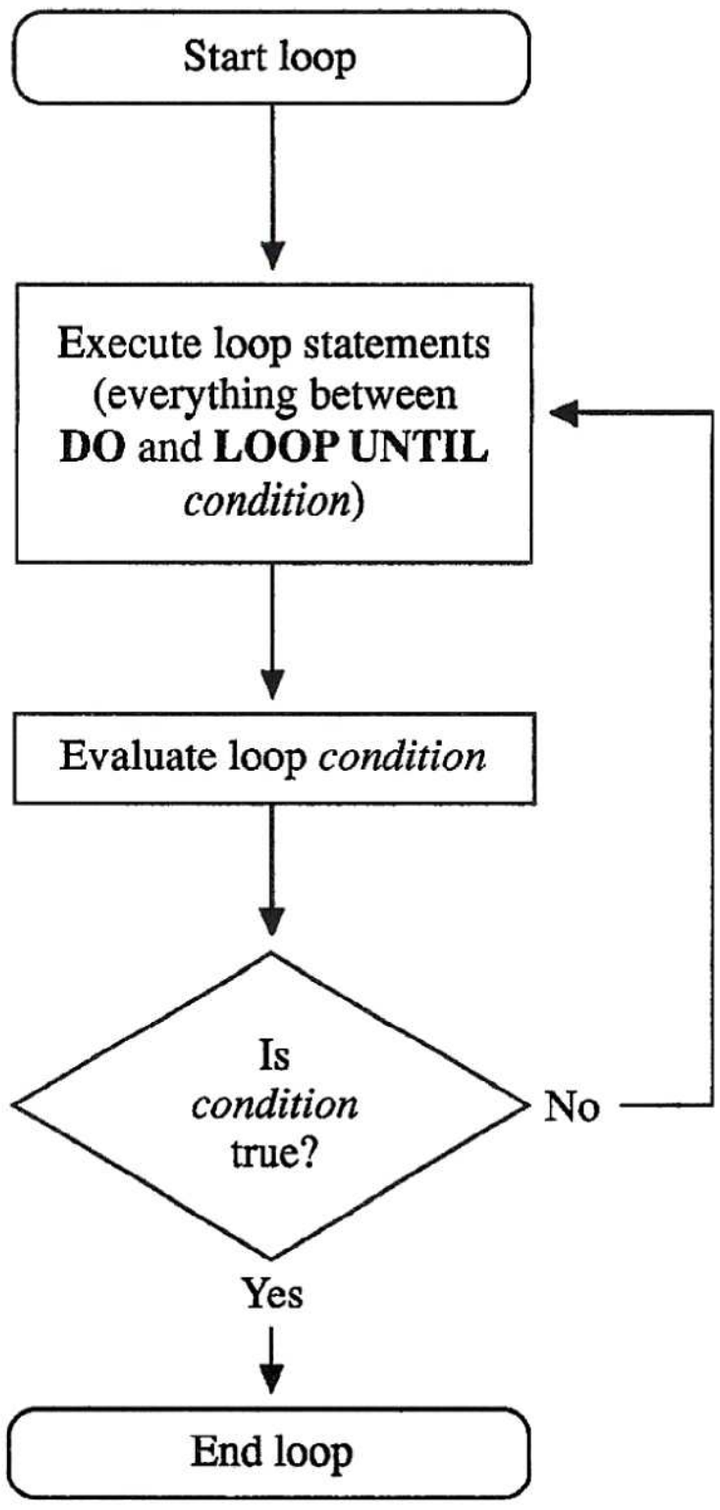


Figure 1.7 Logic of DO...LOOP UNTIL

Loop Tests: One Way to Exit DO...LOOP

You can use a loop test at the end of a **DO...LOOP** statement to create a loop in which the statements always execute at least once. With the **WHILE...WEND** statement, you sometimes have to resort to the trick of presetting the loop variable to some value in order to force the first pass through the loop. With **DO...LOOP**, such tricks are not necessary.

The following examples illustrate both approaches:

```
' WHILE...WEND loop tests at the top, so assigning "Y"
' to Response$ is necessary to force execution of the
' loop at least once:
Response$ = "Y"
WHILE UCASE$(Response$) = "Y"
    .
    .
    .
   INPUT "Do again"; Response$
WEND
' The same loop using DO...LOOP to test after the
' body of the loop:
DO
    .
    .
    .
   INPUT "Do again"; Response$
LOOP WHILE UCASE$(Response$) = "Y"
```

You can also rewrite a condition expressed with **WHILE** using **UNTIL** instead, as in the following:

```
' ========================================================
'                  Using DO WHILE NOT...LOOP
' ========================================================

' While the end of file 1 has not been reached, read
' a line from the file and print it on the screen:
DO WHILE NOT EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT LineBuffer$
LOOP

' --------------------------------------------------------
'                   Using DO UNTIL...LOOP
' ========================================================

' Until the end of file 1 has been reached, read
' a line from the file and print it on the screen:
DO UNTIL EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT LineBuffer$
LOOP
```

EXIT DO: An Alternative Way to Exit DO...LOOP

Inside a **DO...LOOP** statement, other statements are executed that eventually change the loop-test condition from true to false or false to true, ending the loop. In the **DO...LOOP** examples presented so far, the test has occurred either at the beginning or the end of the loop. However, by using the **EXIT DO** statement to exit from the loop, you can move the test elsewhere inside the loop. A single **DO...LOOP** can contain any number of **EXIT DO** statements, and the **EXIT DO** statements can appear anywhere within the loop.

Example The following example opens a file and reads it, one line at a time, until the end of the file is reached or until it finds the pattern entered by the user. If it finds the pattern before getting to the end of the file, an **EXIT DO** statement exits the loop.

```
INPUT "File to search: ", File$
IF File$ = "" THEN END

INPUT "Pattern to search for: ", Pattern$
OPEN File$ FOR INPUT AS #1

DO UNTIL EOF(1)     ' EOF(1) returns a true value if the
                    ' end of the file has been reached.
   LINE INPUT #1, TempLine$
   IF INSTR(TempLine$, Pattern$) > 0 THEN

      ' Print the first line containing the pattern and
      ' exit the loop:
      PRINT TempLine$
      EXIT DO
   END IF
LOOP
```

Sample Applications

The sample applications for this chapter are a checkbook balancing program and a program that ensures that every line in a text file ends with a carriage-return line-feed sequence.

Checkbook Balancing Program (CHECK.BAS)

This program prompts the user for the starting checking account balance and all transactions—withdrawals or deposits—that have occurred. It then prints a sorted list of the transactions and the final balance in the account.

Statements Used

The program demonstrates the following statements discussed in this chapter:

- **DO...LOOP WHILE**
- **FOR...NEXT**
- **EXIT FOR**
- Block **IF...THEN...ELSE**

Program Listing

```
DIM Amount(1 TO 100) AS CURRENCY, Balance AS CURRENCY
CONST FALSE = 0, TRUE = NOT FALSE
CLS
' Get account's starting balance:
INPUT "Type starting balance, then press <ENTER>: ", Balance
' Get transactions. Continue accepting input
' until the input is zero for a transaction,
' or until 100 transactions have been entered:
FOR TransacNum% = 1 TO 100
   PRINT TransacNum%;
   PRINT ") Enter transaction amount (0 to end): ";
   INPUT "", Amount(TransacNum%)
   IF Amount(TransacNum%) = 0 THEN
      TransacNum% = TransacNum% - 1
      EXIT FOR
   END IF
NEXT

' Sort transactions in ascending order,
' using a "bubble sort":
Limit% = TransacNum%
DO
   Swaps% = FALSE
   FOR I% = 1 TO (Limit% - 1)
      ' If two adjacent elements are out of order,
      ' switch those elements:
      IF Amount(I%) > Amount(I% + 1) THEN
         SWAP Amount(I%), Amount(I% + 1)
         Swaps% = I%
      END IF
   NEXT I%
   ' Sort on next pass only to where last switch was made:
   Limit% = Swaps%
' Sort until no elements are exchanged:
LOOP WHILE Swaps%
```

```
' Print the sorted transaction array. If a transaction
' is greater than zero, print it as a "CREDIT"; if a
' transaction is less than zero, print it as a "DEBIT":
FOR I% = 1 TO TransacNum%
   IF Amount(I%) > 0 THEN
      PRINT USING "CREDIT: $$######.##"; Amount(I%)
   ELSEIF Amount(I%) < 0 THEN
      PRINT USING "DEBIT: $$######.##"; Amount(I%)
   END IF
   ' Update balance:
   Balance = Balance + Amount(I%)
NEXT I%
' Print the final balance:
PRINT
PRINT "---------------------------"
PRINT USING "Final Balance: $$######.##"; Balance
END
```

Carriage-Return/Line-Feed Filter (CRLF.BAS)

Some text files are saved in a format that uses only a carriage return (return to the beginning of the line) or a line feed (advance to the next line) to signify the end of a line. Many text editors expand this single carriage return (CR) or line feed (LF) to a carriage-return/line-feed (CR-LF) sequence whenever you load the file for editing. However, if you use a text editor that does not expand a single CR or LF to CR-LF, you may have to modify the file so it has the correct sequence at the end of each line.

The following program is a filter that opens a file, expands a single CR or LF to a CR-LF combination, then writes the adjusted lines to a new file. The original contents of the file are saved in a file with a .BAK extension.

Statements Used

This program demonstrates the following statements discussed in this chapter:

- **DO...LOOP WHILE**
- **DO UNTIL...LOOP**
- Block **IF...THEN...ELSE**
- **SELECT CASE...END SELECT**

To make this program more useful, it contains the following constructions not discussed in this chapter:

- A **FUNCTION** procedure named `Backup$` that creates the file with the .BAK extension.

  See Chapter 2, "SUB and FUNCTION Procedures," for more information on defining and using procedures.

- An error-handling routine named `ErrorHandler` to deal with errors that could occur when the user enters a filename. For instance, if the user enters the name of a nonexistent file, this routine prompts for a new name. Without this routine, such an error would end the program.

  See Chapter 8, "Error Handling," for more information on trapping errors.

Program Listing

```
DEFINT A-Z                  ' Default variable type is integer.

' The Backup$ function makes a backup file with
' the same base as FileName$ plus a .BAK extension:
DECLARE FUNCTION Backup$ (FileName$)

' Initialize symbolic constants and variables:
CONST FALSE = 0, TRUE = NOT FALSE

CarReturn$ = CHR$(13)
LineFeed$ = CHR$(10)

DO
   CLS

   ' Input the name of the file to change:
   INPUT "Which file do you want to convert"; OutFile$

   InFile$ = Backup$(OutFile$)   ' Get backup file's name.

   ON ERROR GOTO ErrorHandler    ' Turn on error trapping.

   NAME OutFile$ AS InFile$      ' Rename input file as
                                 ' backup file.

   ON ERROR GOTO 0               ' Turn off error trapping.

   ' Open backup file for input and old file for output:
   OPEN InFile$ FOR INPUT AS #1
   OPEN OutFile$ FOR OUTPUT AS #2

   ' The PrevCarReturn variable is a flag set to TRUE
   ' whenever the program reads a carriage-return character:
   PrevCarReturn = FALSE
' Read from input file until reaching end of file:
   DO UNTIL EOF(1)
      ' This is not end of file, so read a character:
      FileChar$ = INPUT$(1, #1)
      SELECT CASE FileChar$
```

```
CASE CarReturn$          ' The character is a CR.

   ' If the previous character was also a
   ' CR, put a LF before the character:
   IF PrevCarReturn THEN
  FileChar$ = LineFeed$ + FileChar$
   END IF

   ' In any case, set the PrevCarReturn
   ' variable to TRUE:
   PrevCarReturn = TRUE

CASE LineFeed$           ' The character is a LF.

   ' If the previous character was not a
   ' CR, put a CR before the character:
   IF NOT PrevCarReturn THEN
  FileChar$ = CarReturn$ + FileChar$
   END IF

   ' Set the PrevCarReturn variable to FALSE:
   PrevCarReturn = FALSE

CASE ELSE                ' Neither a CR nor a LF.

   ' If the previous character was a CR,
   ' set the PrevCarReturn variable to FALSE
   ' and put a LF before the current character:
   IF PrevCarReturn THEN
         PrevCarReturn = FALSE
         FileChar$ = LineFeed$ + FileChar$
   END IF

  END SELECT

  ' Write the character(s) to the new file:
     PRINT #2, FileChar$;
  LOOP

  ' Write a LF if the last character in the file was a CR:
  IF PrevCarReturn THEN PRINT #2, LineFeed$;
CLOSE                    ' Close both files.
  PRINT "Another file (Y/N)?"  ' Prompt to continue.

  ' Change the input to uppercase (capital) letter:
  More$ = UCASE$(INPUT$(1))

' Continue the program if the user entered a "Y" or a "y":
LOOP WHILE More$ = "Y"
END
```

```
ErrorHandler:              ' Error-handling routine
   CONST NOFILE = 53, FILEEXISTS = 58

   ' The ERR function returns the error code for last error:
   SELECT CASE ERR
      CASE NOFILE          ' Program couldn't find file
                           ' with input name.
         PRINT "No such file in current directory."
         INPUT "Enter new name: ", OutFile$
         InFile$ = Backup$(OutFile$)
         RESUME
      CASE FILEEXISTS      ' There is already a file named
                           ' <filename>.BAK in this directory:
                           ' remove it, then continue.
         KILL InFile$
         RESUME
      CASE ELSE            ' An unanticipated error occurred:
                           ' stop the program.
         ON ERROR GOTO 0
   END SELECT

' ======================== BACKUP$ ========================
'   This procedure returns a filename that consists of the
'   base name of the input file (everything before the ".")
'   plus the extension ".BAK"
' =========================================================

FUNCTION Backup$ (FileName$) STATIC

   ' Look for a period:
   Extension = INSTR(FileName$, ".")

   ' If there is a period, add .BAK to the base:
   IF Extension > 0 THEN
      Backup$ = LEFT$(FileName$, Extension - 1) + ".BAK"
   ' Otherwise, add .BAK to the whole name:
   ELSE
      Backup$ = FileName$ + ".BAK"
   END IF
END FUNCTION
```

Chapter 2
SUB and FUNCTION Procedures

This chapter explains how to simplify your programming by breaking programs into smaller logical components. These components—known as "procedures"—can then become building blocks that let you enhance and extend the BASIC language itself.

When you are finished with this chapter, you will know how to perform the following tasks with procedures:

- Define and call BASIC procedures.
- Use local and global variables in procedures.
- Use procedures instead of **GOSUB** subroutines and **DEF FN** functions.
- Pass arguments to procedures and return values from procedures.
- Write recursive procedures (procedures that can call themselves).

Although you can create a BASIC program with any text editor, the QuickBASIC Extended (QBX) development environment makes it especially easy to write programs that contain procedures. Also, in most cases QBX automatically generates a **DECLARE** statement when you save your program. (The **DECLARE** statement ensures that only the correct number and type of arguments are passed to a procedure and allows your program to call procedures defined in separate modules.)

Procedures: Building Blocks for Programming

As used in this chapter, the term "procedure" covers **SUB...END SUB** and **FUNCTION...END FUNCTION** constructions. Procedures are useful for condensing repeated tasks. For example, suppose you are writing a program that you intend eventually to compile as a stand-alone application and you want the user of this application to be able to pass several arguments to the application from the command line. It then makes sense to turn this task—breaking the string returned by the **COMMAND$** function into two or more arguments—into a separate procedure. Once you have this procedure up and running, you can use it in other programs. In essence, you are extending BASIC to fit your individual needs when you use procedures.

These are the two major benefits of programming with procedures:

- Procedures allow you to break your programs into discrete logical units, each of which can be more easily debugged than can an entire program without procedures.
- Procedures used in one program can be used as building blocks in other programs, usually with little or no modification.

You can also put procedures in your own Quick library, which is a special file that you can load into memory when you start QBX. Once the contents of a Quick library are in memory with QBX, any program that you write has access to the procedures in the library. This makes it easier for all of your programs to share and save code. (See Chapter 19, "Creating and Using Quick Libraries," for more information on how to build Quick libraries.)

Comparing Procedures with Subroutines

If you are familiar with earlier versions of BASIC, you might think of a **SUB...END SUB** procedure as being roughly similar to a **GOSUB...RETURN** subroutine. You will also notice some similarities between a **FUNCTION...END FUNCTION** procedure and a **DEF FN...END DEF** function. However, procedures have many advantages over these older constructions, as shown in the following sections.

Note To avoid confusion between a **SUB** procedure and the target of a **GOSUB** statement, **SUB** procedures are referred to in this manual as "subprograms," while statement blocks accessed by **GOSUB...RETURN** statements are referred to as "subroutines."

Comparing SUB with GOSUB

Although use of the **GOSUB** subroutine does help break programs into manageable units, **SUB** procedures have a number of advantages over subroutines, as discussed in the following sections.

Local and Global Variables

In **SUB** procedures, all variables are local by default; that is, they exist only within the scope of the **SUB** procedure's definition. To illustrate, the variable named `I%` in the following subprogram is local to the procedure, and has no connection with the variable named `I%` in the module-level code:

```
I% = 1
CALL Test
PRINT I% ' I% still equals 1.
END

SUB Test STATIC
   I% = 50
END SUB
```

A **GOSUB** has a major drawback as a building block in programs: it contains only "global variables." With global variables, if you have a variable named `I%` inside your subroutine, and another variable named `I%` outside the subroutine but in the same module, they are one and the same. Any changes to the value of `I%` in the subroutine affect `I%` everywhere it appears in the module. As a result, if you try to patch a subroutine from one module into another module, you may have to rename subroutine variables to avoid conflict with variable names in the new module.

Use in Multiple-Module Programs

A **SUB** can be defined in one module and called from another. This significantly reduces the amount of code required for a program and increases the ease with which code can be shared among a number of programs.

A **GOSUB** subroutine, however, must be defined and used in the same module.

Operating on Different Sets of Variables

A **SUB** procedure can be called any number of times within a program, with a different set of variables being passed to it each time. This is done by calling the **SUB** procedure with an argument list. (See the section "Passing Arguments to Procedures" later in this chapter for more information on how to do this.) In the following example, the subprogram `Compare` is called twice, with different pairs of variables passed to it each time:

```
X = 4: Y = 5

CALL Compare (X, Y)

Z = 7: W = 2
CALL Compare (Z, W)
END
SUB Compare (A, B)
  IF A < B THEN SWAP A, B
END SUB
```

Calling a **GOSUB** subroutine more than once in the same program and having it operate on a different set of variables each time is difficult. The process involves copying values to and from global variables, as shown in the next example:

```
X = 4: Y = 5
A = X: B = Y
GOSUB Compare
X = A: Y = B

Z = 7 : W = 2
A = Z : B = W
GOSUB Compare
Z = A : W = B
END

Compare:
  IF A < B THEN SWAP A, B
RETURN
```

Comparing FUNCTION with DEF FN

While the multiline **DEF FN** function definition answers the need for functions more complex than can be squeezed onto a single line, **FUNCTION** procedures give you this capability plus the additional advantages discussed in the following sections.

Local and Global Variables

By default, all variables within a **FUNCTION** procedure are local to it, although you do have the option of using global variables. (See the section "Sharing Variables with SHARED" later in this chapter for more information on procedures and global variables.)

In a **DEF FN** function, variables used within the function's body are global to the current module by default (this is also true for **GOSUB** subroutines). However, you can make a variable in a **DEF FN** function local by putting it in a **STATIC** statement.

Changing Variables Passed to the Procedure

Variables are passed to **FUNCTION** procedures by reference or by value. When you pass a variable by reference, you can change the variable by changing its corresponding parameter in the procedure. For example, after the call to `GetRemainder` in the next program, the value of `X` is 2, since the value of `M` is 2 at the end of the procedure:

```
X = 89
Y = 40
PRINT GetRemainder(X, Y)
PRINT X, Y          ' X is now 2.
END
```

```
FUNCTION GetRemainder (M, N)
  GetRemainder = M MOD N
  M = M \ N
END FUNCTION
```

Variables are passed to a **DEF FN** function only by value, so in the next example, `FNRemainder` changes `M` without affecting `X`:

```
DEF FNRemainder (M, N)
  FNRemainder = M MOD N
  M = M \ N
END DEF

X = 89
Y = 40
PRINT FNRemainder(X, Y)

PRINT X,Y   ' X is still 89.
```

See the sections "Passing Arguments by Reference" and "Passing Arguments by Value" later in this chapter for more information on the distinction between passing by reference and by value.

Calling the Procedure Within Its Definition

A **FUNCTION** procedure can be "recursive"; in other words, it can call itself within its own definition. (See the section "Recursive Procedures" later in this chapter for more information on how procedures can be recursive.) A **DEF FN** function cannot be recursive.

Use in Multiple-Module Programs

You can define a **FUNCTION** procedure in one module and use it in another module. You need to put a **DECLARE** statement in the module in which the procedure is used; otherwise, your program thinks the procedure name refers to a variable. (See the section "Checking Arguments with DECLARE" later in this chapter for more information on using **DECLARE** this way.)

A **DEF FN** function can only be used in the module in which it is defined. Unlike **SUB** or **FUNCTION** procedures, which can be called before they appear in the program, a **DEF FN** function must always be defined before it is used in a module.

Note The name of a **FUNCTION** procedure can be any valid BASIC variable name, except one beginning with the letters "FN." The name of a **DEF FN** function must always be preceded by "FN."

Defining Procedures

BASIC procedure definitions have the following general syntax:

{SUB | FUNCTION} *procedurename* **[[**(*parameterlist*)**]] [[STATIC]]**
[[*statementblock-1* **]]**
[[EXIT {SUB | FUNCTION}
[[*statementblock-2***]]]]**
END {SUB | FUNCTION}

The following table describes the parts of a procedure definition:

Part	Description
{SUB | FUNCTION}	Marks the beginning of a **SUB** or **FUNCTION** procedure, respectively.
procedurename	Any valid variable name up to 40 characters long. The same name cannot be used for a **SUB** and a **FUNCTION** procedure.
parameterlist	A list of variables, separated by commas, that shows the number and type of arguments to be passed to the procedure. (The section "Parameters and Arguments" later in this chapter explains the difference between parameters and arguments.)
STATIC	If you use the **STATIC** attribute, local variables are static; that is, they retain their values between calls to the procedure. If you omit the **STATIC** attribute, local variables are "automatic" by default; that is, they are initialized to zeros or null strings at the start of each procedure call. See the section "Automatic and Static Variables" later in this chapter for more information.

END {SUB | FUNCTION}

Ends a **SUB** or **FUNCTION** definition. To run correctly, every procedure must have exactly one **END {SUB | FUNCTION}** statement.

When your program encounters an **END SUB** or **END FUNCTION**, it exits the procedure and returns to the statement immediately following the one that called the procedure. You can also use one or more optional **EXIT {SUB | FUNCTION}** statements within the body of a procedure definition to exit from the procedure.

All valid BASIC expressions and statements except the following are allowed within a procedure definition.

- **DEF FN...END DEF, FUNCTION...END FUNCTION,** or **SUB...END SUB**
  It is not possible to nest procedure definitions or to define a **DEF FN** function inside a procedure. However, a procedure can call another procedure or a **DEF FN** function.
- **COMMON.**
- **DECLARE.**
- **DIM SHARED.**
- **OPTION BASE.**
- **TYPE...END TYPE.**

Example The following example is a **FUNCTION** procedure named `IntegerPower`:

```
FUNCTION IntegerPower& (X&, Y&) STATIC
  PowerVal& = 1
  FOR I& = 1 TO Y&
   PowerVal& = PowerVal& * X&
  NEXT I&
  IntegerPower& = PowerVal&
END FUNCTION
```

Calling Procedures

Calling a **FUNCTION** procedure is different from calling a **SUB** procedure, as shown in the next two sections.

Calling a FUNCTION Procedure

You call a **FUNCTION** procedure the same way you use an intrinsic BASIC function such as **ABS**, that is, by using its name in an expression. For example, any of the following statements would call a **FUNCTION** named `ToDec`:

```
PRINT 10 * ToDec
X = ToDec
IF ToDec = 10 THEN PRINT "Out of range."
```

A **FUNCTION** procedure can return values by changing variables passed to it as arguments. (See the section "Passing Arguments by Reference" later in this chapter for an explanation of how this is done.) Additionally, a **FUNCTION** procedure returns a single value in its name, so the name of the function must agree with the type it returns. For example, if the function returns a string value, either its name must have the string type-declaration character (**$**) appended to it, or it must be declared as having the string type in a preceding **DEFSTR** statement.

Example

The following program shows a **FUNCTION** procedure that returns a string value. Note that the type-declaration suffix for strings (**$**) is part of the procedure name.

```
DECLARE FUNCTION GetInput$ ()
Banner$ = GetInput$     ' Call the function and assign the
                        ' return value to a string variable.
PRINT Banner$           ' Print the string.
END

' ======================== GetInput$ ========================
'    The $ type-declaration character at the end of this
'    function name means that it returns a string value.
' ===========================================================

FUNCTION GetInput$ STATIC

   ' Return a string of 10 characters read from the
   ' keyboard, echoing each character as it is typed:
   FOR I% = 1 TO 10
      Char$ = INPUT$(1)      ' Get the character.
      PRINT Char$;           ' Echo the character on the
                             ' screen.
      Temp$ = Temp$ + Char$ ' Add the character to the
                             ' string.
```

```
    NEXT
    PRINT
    GetInput$ = Temp$     ' Assign the string to the procedure.
END FUNCTION
```

Calling a SUB Procedure

A **SUB** procedure differs from a **FUNCTION** procedure in that a **SUB** procedure cannot be called by using its name within an expression. A call to a **SUB** procedure is a stand-alone statement, like BASIC's **CIRCLE** statement. Also, a **SUB** procedure does not return a value in its name as does a function. However, like a function, a **SUB** procedure can modify the values of any variables passed to it. (The section "Passing Arguments by Reference" later in this chapter explains how this is done.)

There are two ways to call a **SUB** procedure:

- Put its name in a **CALL** statement:

  ```
  CALL PrintMessage
  ```

- Use its name as a statement by itself:

  ```
  PrintMessage
  ```

If you omit the **CALL** keyword, don't put parentheses around arguments passed to the **SUB** procedure:

```
' Call the ProcessInput subprogram with CALL and pass the
' three arguments First$, Second$, and NumArgs% to it:
CALL ProcessInput (First$, Second$, NumArgs%)

' Call the ProcessInput subprogram without CALL and pass
' it the same arguments (note no parentheses around the
' argument list):
ProcessInput First$, Second$, NumArgs%
```

See the next section for more information on passing arguments to procedures.

If your program calls **SUB** procedures without using **CALL**, and if you are not using QBX to write the program, you must put the name of the subprogram in a **DECLARE** statement before it is called:

```
DECLARE SUB CheckForKey
.
.
.
CheckForKey
```

You need to be concerned about this only if you are developing programs outside QBX, since QBX automatically inserts **DECLARE** statements wherever they are needed when it saves a program.

Passing Arguments to Procedures

The following sections explain how to tell the difference between parameters and arguments, how to pass arguments to procedures, and how to check arguments to make sure they are of the correct type and quantity.

Parameters and Arguments

The first step in learning about passing arguments to procedures is understanding the difference between the terms "parameter" and "argument":

Parameter	Argument
A variable name that appears in a **SUB**, **FUNCTION**, or **DECLARE** statement	A constant, variable, or expression passed to a **SUB** or **FUNCTION** when the procedure is called

In a procedure definition, parameters are placeholders for arguments. As shown in Figure 2.1, when a procedure is called, arguments are plugged into the variables in the parameter list, with the first parameter receiving the first argument, the second parameter receiving the second argument, and so on.

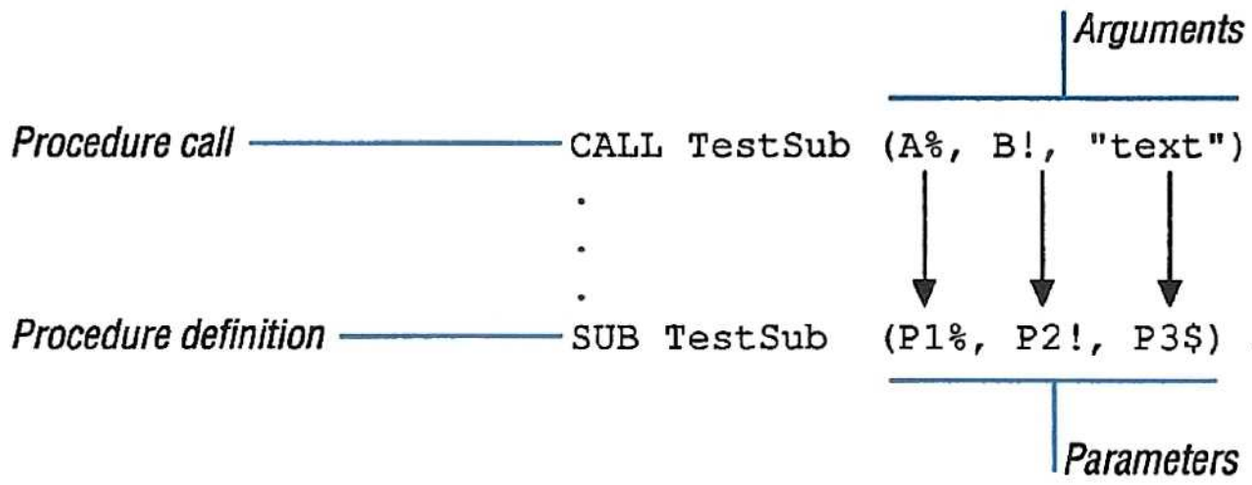


Figure 2.1 Parameters and Arguments

Figure 2.1 also demonstrates another important rule: although the names of variables do not have to be the same in an argument list and a parameter list, the number of parameters and the number of arguments do. Furthermore, the type (string, integer numeric, single-precision numeric, and so on) should be the same for corresponding arguments and parameters. (See the section "Checking Arguments with DECLARE" later in this chapter for more information on how to ensure that arguments and parameters agree in number and type.)

A parameter list consists of any of the following, all separated by commas:

- Valid variable names, except for fixed-length strings.

  For example, `X$` and `X AS STRING` are both legal in a parameter list, since they refer to variable-length strings. However, `X AS STRING * 10` refers to a fixed-length string 10 characters long and cannot appear in a parameter list. (Fixed-length strings are perfectly all right as *arguments* passed to procedures. See Chapter 4, "String Processing," for more information on fixed-length and variable-length strings.)

- Array names followed by a pair of left and right parentheses.

An argument list consists of any of the following, all separated by commas:

- Constants.
- Expressions.
- Valid variable names.
- Array names followed by left and right parentheses.

Examples The following example shows the first line of a subprogram definition with a parameter list:

```
SUB TestSub (A%, Array(), RecVar AS RecType, Cs$)
```

The first parameter, `A%`, is an integer; the second parameter, `Array()`, is a single-precision array, since untyped numeric variables are single precision by default; the third parameter, `RecVar`, is a record of type `RecType`; and the fourth parameter, `Cs$`, is a string.

The `CALL TestSub` line in the next example calls the `TestSub` subprogram and passes it four arguments of the appropriate type:

```
TYPE RecType
  Rank AS STRING * 12
  SerialNum AS LONG
END TYPE

DIM RecVar AS RecType

CALL TestSub (X%, A(), RecVar, "Daphne")
```

Passing Constants and Expressions

Constants—whether string or numeric—can appear in the list of arguments passed to a procedure. Naturally, a string constant must be passed to a string parameter and a numeric constant to a numeric parameter, as shown in the next example:

```
CONST SCREENWIDTH = 80
CALL PrintBanner (SCREENWIDTH, "Monthly Status Report")
.
.
.
SUB PrintBanner (SW%, Title$)
   .
   .
   .
END SUB
```

If a numeric constant in an argument list does not have the same type as the corresponding parameter in the **SUB** or **FUNCTION** statement, then the constant is coerced to the type of the parameter, as you can see by the output from the next example:

```
CALL test(4.6, 4.1)
END

SUB test (x%, y%)
  PRINT x%, y%
END SUB
```

Output

```
5         4
```

Expressions resulting from operations on variables and constants can also be passed to a procedure. As is the case with constants, numeric expressions that disagree in type with their parameters are coerced into agreement, as shown here:

```
Checker A! + 25!, NOT BooleanVal%

' In the next call, putting parentheses around the
' long-integer variable Bval& makes it an expression.
' The (Bval&) expression is coerced to a short integer
' in the Checker procedure:
Checker A! / 3.1, (Bval&)
.
.
.
END

SUB Checker (Param1!, Param2%)
   .
   .
   .
END SUB
```

Passing Variables

This section discusses how to pass simple variables, complete arrays, elements of arrays, records, and elements of records to procedures.

Passing Simple Variables

In argument and parameter lists, you can declare the type for a simple variable in one of the following three ways:

- Append one of the type-declaration suffixes (**%**, **&**, **!**, **#**, **@** or **$**) to the variable name.
- Declare the variable in a **DIM**, **COMMON**, **REDIM**, **SHARED**, or **STATIC** statement. For example:

```
DIM A AS LONG
```

- Use a **DEF***type* statement to set the default type.

No matter which method you choose, corresponding variables must have the same type in the argument and parameter lists, as shown in the example below.

Example

In this example, two arguments are passed to the **FUNCTION** procedure. The first is an integer giving the length of the string returned by `CharString$`, while the second is a character that is repeated to make the string.

```
FUNCTION CharString$(A AS INTEGER, B$) STATIC
  CharString$ = STRING$(A%, B$)
END FUNCTION

DIM X AS INTEGER
INPUT "Enter a number (1 to 80): ", X
INPUT "Enter a character: ", Y$

' Print a string consisting of the Y$ character, repeated
' X number of times:
PRINT CharString$(X, Y$)
END
```

Output

```
Enter a number (1 to 80): 21
Enter a character: #
#####################
```

Passing an Entire Array

To pass all the elements of an array to a procedure, put the array's name, followed by left and right parentheses, in the argument and parameter lists.

Example This example shows how to pass all the elements of an array to a procedure:

```
DIM Values(1 TO 5) AS INTEGER

' Note empty parentheses after array name when calling
' procedure and passing array:
CALL ChangeArray (1, 5, Values())
CALL PrintArray (1, 5, Values())
END

' Note empty parentheses after P parameter:

SUB ChangeArray (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      P(I%) = I% ^ 3
   NEXT I%
END SUB

SUB PrintArray (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      PRINT P(I%)
   NEXT I%
   PRINT
END SUB
```

Passing Individual Array Elements

If a procedure does not require an entire array, you can pass individual elements of the array instead. To pass an element of an array, use the array name followed by the appropriate subscripts inside parentheses.

Example The `SqrVal Array(4,2)` statement in the following example passes the element in row 4, column 2 of the array to the `SqrVal` subprogram. (Note how the subprogram actually changes the value of this array element.)

```
DIM Array(1 TO 5,1 TO 3)

Array(4,2) = -36
PRINT Array(4,2)
SqrVal Array(4,2)
PRINT Array(4,2)        ' The call to SqrVal has changed
                        ' the value of Array(4,2).
END
SUB SqrVal(A) STATIC
  A = SQR(ABS(A))
END SUB
```

Output

```
-36
 6
```

Using Array-Bound Functions

The **LBOUND** and **UBOUND** functions provide a useful way to determine the size of an array passed to a procedure. The **LBOUND** function finds the smallest index value of an array subscript, while the **UBOUND** function finds the largest one. These functions save you the trouble of having to pass the upper and lower bounds of each array dimension to a procedure.

Example The subprogram in the following example uses the **LBOUND** function to initialize the variables `Row` and `Col` to the lowest subscript values in each dimension of `A`. It also uses the **UBOUND** function to limit the number of times the **FOR** loop executes to the number of elements in the array.

```
SUB PrintOut(A()) STATIC
  FOR Row = LBOUND(A,1) TO UBOUND(A,1)
   FOR Col = LBOUND(A,2) TO UBOUND(A,2)
   PRINT A(Row,Col)
   NEXT Col
  NEXT Row
END SUB
```

Passing an Entire Record

To pass a complete record (a variable declared as having a user-defined type) to a procedure, complete the following steps:

1. Define the type (`StockItem` in this example).

```
TYPE StockItem
  PartNumber AS STRING * 6
  Description AS STRING * 20
  UnitPrice  AS SINGLE
  Quantity  AS INTEGER
END TYPE
```

2. Declare a variable (`StockRecord`) with that type.

```
DIM StockRecord AS StockItem
```

3. Call a procedure (`FindRecord`) and pass it the variable you have declared.

```
CALL FindRecord(StockRecord)
```

4. In the procedure definition, give the parameter the same type as the variable.

```
SUB FindRecord (RecordVar AS StockItem) STATIC
.
.
.
END SUB
```

Passing Individual Elements of a Record

To pass an individual element in a record to a procedure, put the name of the element (*recordname.elementname*) in the argument list. Be sure, as always, that the corresponding parameter in the procedure definition agrees with the type of that element.

Example The following example shows how to pass the two elements in the record variable `StockItem` to the `PrintTag` **SUB** procedure. Note how each parameter in the **SUB** procedure agrees with the type of the individual record elements.

```
TYPE StockType
  PartNumber AS STRING * 6
  Descrip AS STRING * 20
  UnitPrice AS CURRENCY
  Quantity AS INTEGER
END TYPE

DIM StockItem AS StockType

CALL PrintTag (StockItem.Descrip, StockItem.UnitPrice)
.
.
.
END

SUB PrintTag (Desc$, Price AS CURRENCY)
   .
   .
   .
END SUB
```

Checking Arguments with DECLARE

If you are using QBX to write your program, you will notice that QBX automatically inserts a **DECLARE** statement for each procedure whenever you save the program. Each **DECLARE** statement consists of the word **DECLARE**, followed by the words **SUB** or **FUNCTION**, the name of the procedure, and a set of parentheses. If the procedure has no parameters, then the parentheses are empty. If the procedure has parameters, then the parentheses enclose a parameter list that specifies the number and type of the arguments to be passed to the procedure. This parameter list has the same format as the list in the definition line found in the **SUB** or **FUNCTION** procedure.

The purpose of the parameter list in a **DECLARE** statement is to turn on "type checking" of arguments passed to the procedure. That is, every time the procedure is called with variable arguments, those variables are checked to be sure they agree with the number and type of the parameters in the **DECLARE** statement.

QBX puts all procedure definitions at the end of a module when it saves a program. Therefore, if there are no **DECLARE** statements, when you try to compile a program with the BC command you would run into a problem known as "forward reference" (calling a procedure before it is defined). By generating a prototype of the procedure definition, **DECLARE** statements allow your program to call procedures that are defined later in a module, or in another module altogether.

Examples

The next example shows an empty parameter list in the **DECLARE** statement, since no arguments are passed to `GetInput$`:

```
DECLARE FUNCTION GetInput$ ()
X$ = GetInput$

FUNCTION GetInput$ STATIC
  GetInput$ = INPUT$(10)
END FUNCTION
```

The next example shows a parameter list in the **DECLARE** statement, since an integer argument is passed to this version of `GetInput$`:

```
DECLARE FUNCTION GetInput$ (X%)
X$ = GetInput$ (5)

FUNCTION GetInput$ (X%) STATIC
  GetInput$ = INPUT$(X%)
END FUNCTION
```

When QBX Does Not Generate a DECLARE Statement

In certain instances, QBX does not generate **DECLARE** statements in the module that calls a procedure.

QBX cannot generate a **DECLARE** statement in one module for a **SUB** procedure defined in another module if the module containing the definition is not loaded. However, the **DECLARE** statement is not needed unless you want to call the **SUB** procedure without using the keyword **CALL**.

QBX does not generate a **DECLARE** statement for a **FUNCTION** procedure, whether that module is loaded or not. In such a case, you must type the **DECLARE** statement yourself at the beginning of the module where the **FUNCTION** procedure is called; otherwise, QBX considers the call to the procedure to be a variable name.

QBX also cannot generate a **DECLARE** statement for any procedure in a Quick library. You must add one to the program yourself.

Developing Programs Outside the QBX Environment

If you are writing your programs with your own text editor and then compiling them outside the QBX environment with the BC and LINK commands, be sure to put **DECLARE** statements in the following three locations:

- At the beginning of any module that calls a **FUNCTION** procedure before it is defined:

```
DECLARE FUNCTION Hypot (X!, Y!)

INPUT X, Y
PRINT Hypot(X, Y)
END

FUNCTION Hypot (A, B) STATIC
  Hypot = SQR(A ^ 2 + B ^ 2)
END FUNCTION
```

- At the beginning of any module that calls a **SUB** procedure before it is defined and does not use **CALL** when calling the **SUB** procedure:

```
DECLARE SUB PrintString (X, Y)
INPUT X, Y

PrintString X, Y    ' Note: no parentheses around
                    ' arguments
END

SUB PrintString (A,B) STATIC

  ' Convert the numbers to strings, remove any leading
  ' blanks from the second number, and print:
  PRINT STR$(A) + LTRIM$(STR$(B))
END SUB
```

When you call a **SUB** procedure with **CALL**, you don't have to declare the **SUB** procedure first:

```
A$ = "466"
B$ = "123"
CALL PrintString(A$, B$)
END

SUB PrintString (X$, Y$) STATIC
  PRINT VAL(X$) + VAL(Y$)
END SUB
```

- At the beginning of any module that calls a **SUB** or **FUNCTION** procedure defined in another module (an "external procedure").

If your procedure has no parameters, remember to put empty parentheses after the name of the procedure in the **DECLARE** statement, as in the next example:

```
DECLARE FUNCTION GetHour$ ()
PRINT GetHour$
END

FUNCTION GetHour$ STATIC
  GetHour$ = LEFT$(TIME$,2)
END FUNCTION
```

Remember, a **DECLARE** statement can appear only at the module level, not the procedure level. A **DECLARE** statement affects the entire module in which it appears.

Using Include Files for Declarations

If you have created a separate procedure-definition module that defines one or more **SUB** or **FUNCTION** procedures, it is a good idea to make an include file to go along with this module. This include file should contain the following:

- **DECLARE** statements for all the module's procedures.
- **TYPE...END TYPE** record definitions for any record parameters in this module's **SUB** or **FUNCTION** procedures.
- **COMMON** statements listing variables shared between this module and other modules in the program. (See the section "Sharing Variables with Other Modules" later in this chapter for more information on using **COMMON** for this purpose.)

Every time you use the definition module in one of your programs, insert a **$INCLUDE** metacommand at the beginning of any module that invokes procedures in the definition module. When your program is compiled, the actual contents of the include file are substituted for the **$INCLUDE** metacommand.

A simple rule of thumb is to make an include file for every module and then use the module and the include file together as outlined previously. The following list itemizes some of the benefits of this technique:

- A module containing procedure definitions remains truly modular—that is, you don't have to copy all the **DECLARE** statements for its procedures every time you call them from another module; instead, you can just substitute one **$INCLUDE** metacommand.
- In QBX, using an include file for procedure declarations suppresses automatic generation of **DECLARE** statements when you save a program.
- Using an include file for declarations avoids problems with getting one module to recognize a **FUNCTION** procedure in another module. (See the section "When QBX Does Not Generate a DECLARE Statement" earlier in this chapter for more information.)

You can take advantage of QBX's facility for generating **DECLARE** statements when creating your include file. The following steps show you how to do this:

1. Create your module.
2. Within that module, call any **SUB** or **FUNCTION** procedures you have defined.
3. Save the module to get automatic **DECLARE** statements for all the procedures.
4. Re-edit the module, removing the procedure calls and moving the **DECLARE** statements to a separate include file.

See the *BASIC Language Reference* for more information on the syntax and usage of the **$INCLUDE** metacommand.

Example The following fragments illustrate how to use a definition module and an include file together:

```
' ========================================================
'                        MODDEF.BAS
' This module contains definitions for the Prompter and
' Max! procedures.
' ========================================================
'
FUNCTION Max! (X!, Y!) STATIC
   IF X! > Y! THEN Max! = X! ELSE Max! = Y!
END FUNCTION
SUB Prompter (Row%, Column%, RecVar AS RecType) STATIC
   LOCATE Row%, Column%
   INPUT "Description: ", RecVar.Description
   INPUT "Quantity:    ", RecVar.Quantity
END SUB

' ========================================================
'                        MODDEF.BI
'  This is an include file that contains DECLARE statements
'  for the Prompter and Max! procedures (as well as a TYPE
'  statement defining the RecType user type). Use this file
'  whenever you use the MODDEF.BAS module.
' ========================================================
'
TYPE RecType
   Description AS STRING * 15
   Quantity AS INTEGER
END TYPE

DECLARE FUNCTION Max! (X!, Y!)
DECLARE SUB Prompter (Row%, Column%, RecVar AS RecType)
```

```
' ============================================================
'                    SAMPLE.BAS
'   This module is linked with the MODDEF.BAS module, and
'   calls the Prompter and Max! procedures in MODDEF.BAS.
' ============================================================
'
' The next line makes the contents of the MODDEF.BI include
' file part of this module as well:
' $INCLUDE: 'MODDEF.BI'
.
.
.
INPUT A, B
PRINT Max!(A, B)        ' Call the Max! FUNCTION procedure in MODDEF.BAS.
.
.
.
Prompter 5, 5, RecVar  ' Call the Prompter SUB procedure in MODDEF.BAS
.
.
.
```

Important While it is good programming practice to put procedure declarations in an include file, do not put the procedures themselves (**SUB...END SUB** or **FUNCTION...END FUNCTION** blocks) in an include file. Procedure definitions are not allowed inside include files in QBX. If you have used include files to define **SUB** procedures in programs written with QuickBASIC versions 2.0 or 3.0, either put these definitions in a separate module or incorporate them into the module where they are called.

Declaring Procedures in Quick Libraries

A convenient programming practice is to put all the declarations for procedures in a Quick library into one include file. With the **$INCLUDE** metacommand you can then incorporate this include file into programs using the library. This saves you the trouble of copying all the relevant **DECLARE** statements every time you use the library.

Passing Arguments by Reference

By default, variables—whether simple scalar variables, arrays and array elements, or records—are passed "by reference" to **FUNCTION** and **SUB** procedures. Here is what is meant by passing variables by reference:

- Each program variable has an address or a location in memory where its value is stored.
- The process of calling a procedure and passing variables to it by reference calls the procedure and passes it the address of each variable. This means that the address of the variable and the address of its corresponding parameter in the procedure are one and the same.

- Therefore, if the procedure modifies the value of the parameter, it also modifies the value of the variable that is passed.

If you do not want a procedure to change the value of a variable, pass the procedure the value contained in the variable, not the address. This way, changes are made only to a copy of the variable, not the variable itself. See the next section for a discussion of this alternative way of passing variables.

Example In the following program, changes made to the parameter `A$` in the `Replace` procedure also change the argument `Test$`:

```
Test$ = "a string with all lowercase letters."
PRINT "Before subprogram call: "; Test$
CALL Replace (Test$, "a")
PRINT "After subprogram call: "; Test$
END

SUB Replace (A$, B$) STATIC
   Start = 1
   DO

      ' Look for B$ in A$, starting at the character
      ' with position "Start" in A$:
      Found = INSTR(Start, A$, B$)
     ' Make every occurrence of B$ in A$
     ' an uppercase letter:
     IF Found > 0 THEN
    MID$(A$,Found) = UCASE$(B$)
    Start = Start + 1
     END IF
  LOOP WHILE Found > 0
END SUB
```

Output

```
Before subprogram call: a string with all lowercase letters.
After subprogram call: A string with All lowercAse letters.
```

Passing Arguments by Value

Passing an argument "by value" means the value of the argument is passed, rather than its address. This prevents the original variable from being changed by the procedure that is called. In BASIC procedures, an actual value cannot be passed, but the same result is achieved by putting parentheses around the variable name. This causes BASIC to treat the variable as an expression. In this case, as with all expressions, the variable is copied to a temporary location, and the address of this temporary location is passed. Since the procedure does not have access to the address of the original variable, it cannot change the original variable; it makes all changes to the copy instead.

Expressions are passed to procedures as in the following:

```
' A + B is an expression; the values of A and B
' are not affected by this procedure call:
CALL Mult(A + B, C)
```

Any variable enclosed in parentheses is treated by BASIC as an expression, as shown in the next example.

Example

In this example, a variable is enclosed in parentheses and passed to a procedure. This simulates actual passing by value because the variable data is copied to a temporary location whose address is passed. As you can see from the output that follows, changes to the **SUB** procedure's local variable `Y` are passed back to the module-level code as changes to the variable `B`. However, changes to `X` in the procedure do not affect the value of `A`, since `A` is passed by value.

```
A = 1
B = 1
PRINT "Before subprogram call, A ="; A; ", B ="; B

' A is passed by value, and B is passed by reference:
CALL Mult((A), B)
PRINT "After subprogram call, A ="; A; ", B ="; B
END
SUB Mult (X, Y) STATIC
  X = 2 * X
  Y = 3 * Y
  PRINT "In subprogram, X ="; X; ", Y ="; Y
END SUB
```

Output

```
Before subprogram call, A = 1 , B = 1
In subprogram, X = 2 , Y = 3
After subprogram call, A = 1 , B = 3
```

Sharing Variables with SHARED

In addition to passing variables through argument and parameter lists, procedures can also share variables with other procedures and with code at the module level (that is, code within a module but outside of any procedure) in one of the following two ways:

- Variables listed in a **SHARED** statement within a procedure are shared only between that procedure and the module-level code. Use this method when different procedures in the same module need different combinations of module-level variables.

- Variables listed in a module-level **COMMON SHARED**, **DIM SHARED**, or **REDIM SHARED** statement are shared between the module-level code and all procedures within that module. This method is most useful when all procedures in a module use a common set of variables.

You can also use the **COMMON** or **COMMON SHARED** statement to share variables among two or more modules. The next three sections discuss these three ways to share variables.

Sharing Variables with Specific Procedures in a Module

If different procedures within a module need to share different variables with the module-level code, use the **SHARED** statement within each procedure.

Arrays in **SHARED** statements consist of the array name followed by a set of empty parentheses:

```
SUB JustAnotherSub STATIC
  SHARED ArrayName ()
  .
  .
  .
```

If you give a variable its type in an **AS** *type* clause, then the variable must also be typed with the **AS** *type* clause in a **SHARED** statement:

```
DIM Buffer AS STRING * 10
.
.
.
END

SUB ReadRecords STATIC
  SHARED Buffer AS STRING * 10
  .
  .
  .
END SUB
```

Example In this example, the **SHARED** statements in the `GetRecords` and `InventoryTotal` procedures show the format of a shared variable list:

```
DECLARE SUB GetRecords ()
DECLARE FUNCTION InventoryTotal! ()
' ========================================================
'                      MODULE-LEVEL CODE
' ========================================================
TYPE RecType
   Price AS SINGLE
   Desc AS STRING * 35
END TYPE

DIM RecVar(1 TO 100) AS RecType     ' Array of records

INPUT "File name: ", FileSpec$
CALL GetRecords
PRINT InventoryTotal
END

' ========================================================
'                     PROCEDURE-LEVEL CODE
' ========================================================
SUB GetRecords STATIC

' Both FileSpec$ and the RecVar array of records
' are shared with the module-level code above:
   SHARED FileSpec$, RecVar() AS RecType
   OPEN FileSpec$ FOR RANDOM AS #1
    .
    .
    .
END SUB

FUNCTION InventoryTotal STATIC

' Only the RecVar array is shared with the module-level
' code:
   SHARED RecVar() AS RecType
    .
    .
    .
END FUNCTION
```

Sharing Variables with All Procedures in a Module

If variables are declared at the module level with the **SHARED** attribute in a **COMMON**, **DIM**, or **REDIM** statement (for example, by using a statement of the form **COMMON SHARED** *variablelist*), then all procedures within that module have access to those variables; in other words, the **SHARED** attribute makes variables global throughout a module.

The **SHARED** attribute is convenient when you need to share large numbers of variables among all procedures in a module.

Examples

These statements declare variables shared among all procedures in one module:

```
COMMON SHARED A, B, C
DIM SHARED Array(1 TO 10, 1 TO 10) AS UserType
REDIM SHARED Alpha(N%)
```

In the following example, the module-level code shares the string array `StrArray` and the integer variables `Min` and `Max` with the two **SUB** procedures `FillArray` and `PrintArray`:

```
' ========================================================
'                     MODULE-LEVEL CODE
' ========================================================
'
DECLARE SUB FillArray ()
DECLARE SUB PrintArray ()

' The following DIM statements share the Min and Max
' integer variables and the StrArray string array
' with any SUB or FUNCTION procedure in this module:
DIM SHARED StrArray (33 TO 126) AS STRING * 5
DIM SHARED Min AS INTEGER, Max AS INTEGER

Min = LBOUND(StrArray)
Max = UBOUND(StrArray)

FillArray       ' Note the absence of argument lists.
PrintArray
END

' ========================================================
'                   PROCEDURE-LEVEL CODE
' ========================================================
'
SUB FillArray STATIC

   ' Load each element of the array from 33 to 126
   ' with a 5-character string, each character of which
   ' has the ASCII code I%:
   FOR I% = Min TO Max
```

```
        StrArray(I%) = STRING$(5, I%)
    NEXT

END SUB

SUB PrintArray STATIC
    FOR I% = Min TO Max
        PRINT StrArray(I%)
    NEXT
END SUB
```

Partial Output

```
!!!!!
"""""
#####
$$$$$
%%%%%
&&&&&
'''''
.
.
.
```

If you are using your own text editor to write your programs and directly compiling those programs outside the QBX development environment, note that variable declarations with the **SHARED** attribute must precede the procedure definition. Otherwise, the value of any variable declared with **SHARED** is not available to the procedure, as shown by the output from the next example. (If you are using QBX to create your programs, this sequence is not required, since QBX automatically saves programs in the correct order.)

```
DEFINT A-Z

FUNCTION Adder (X, Y) STATIC
  Adder = X + Y + Z
END FUNCTION

DIM SHARED Z
Z = 2
PRINT Adder (1, 3)
END
```

Output

```
4
```

The next example shows how you should save the module shown previously, with the definition of `Adder` following the `DIM SHARED Z` statement:

```
DEFINT A-Z

DECLARE FUNCTION Adder (X, Y)

' The variable Z is now shared with Adder:
DIM SHARED Z
Z = 2
PRINT Adder (1, 3)
END

FUNCTION Adder (X, Y) STATIC
  Adder = X + Y + Z
END FUNCTION
```

Output

```
6
```

Sharing Variables with Other Modules

If you want to share variables across modules in your program, list the variables in **COMMON** or **COMMON SHARED** statements at the module level in each module.

Examples The following example shows how to share variables between modules by using a **COMMON** statement in the module that calls the **SUB** procedures, as well as a **COMMON SHARED** statement in the module that defines the procedures. With **COMMON SHARED**, all procedures in the second module have access to the common variables.

```
' ========================================================
'                        MAIN MODULE
' ========================================================

COMMON A, B
A = 2.5
B = 1.2
CALL Square
CALL Cube
END
```

```
' ============================================================
'          Module with Cube and Square Procedures
' ============================================================

' NOTE: The names of the variables (X, Y) do not have to be
' the same as in the other module (A, B). Only the types
' have to be the same.

COMMON SHARED X, Y      ' This statement is at the module level.
                        ' Both X and Y are shared with the CUBE
                        ' and SQUARE procedures below.
SUB Cube STATIC
   PRINT "A cubed   ="; X ^ 3
   PRINT "B cubed   ="; Y ^ 3
END SUB

SUB Square STATIC
   PRINT "A squared ="; X ^ 2
   PRINT "B squared ="; Y ^ 2
END SUB
```

The following example uses named **COMMON** blocks at the module levels and **SHARED** statements within procedures to share different sets of variables with each procedure:

```
DECLARE SUB VolumeCalc ()
DECLARE SUB DensityCalc ()
' ============================================================
'                          MAIN MODULE
' Prints the volume and density of a filled cylinder given
' the input values.
' ============================================================

COMMON /VolumeValues/ Height, Radius, Volume
COMMON /DensityValues/ Weight, Density

INPUT "Height of cylinder in centimeters: ", Height
INPUT "Radius of cylinder in centimeters: ", Radius
INPUT "Weight of filled cylinder in grams: ", Weight

CALL VolumeCalc
CALL DensityCalc

PRINT "Volume is"; Volume; "cubic centimeters."
PRINT "Density is"; Density; "grams/cubic centimeter."
END
```

```
' ============================================================
'     Module with DensityCalc and VolumeCalc Procedures
' ============================================================

COMMON /VolumeValues/ H, R, V
COMMON /DensityValues/ W, D

SUB DensityCalc STATIC

   ' Share the Weight, Volume, and Density variables
   ' with this procedure:
   SHARED W, V, D
   D = W / V
END SUB

SUB VolumeCalc STATIC

   ' Share the Height, Radius, and Volume variables
   ' with this procedure:
   SHARED H, R, V
   CONST PI = 3.141592653589#
   V = PI * H * (R ^ 2)
END SUB
```

Output

```
Height of cylinder in centimeters: 100
Radius of cylinder in centimeters: 10
Weight of filled cylinder in grams: 10000
Volume is 31415.93 cubic centimeters.
Density is .3183099 grams/cubic centimeter.
```

The Problem of Variable Aliases

"Variable aliases" can become a problem in long programs containing many variables and procedures. Variable aliases occur when two or more names refer to the same location in memory. Situations where it arises are:

- When the same variable appears more than once in the list of arguments passed to a procedure.
- When a variable passed in an argument list is also accessed by the procedure by means of the **SHARED** statement or the **SHARED** attribute.

To avoid alias problems, double-check variables shared with a procedure to make sure they don't also appear in a procedure call's argument list. Also, don't pass the same variable twice, as in the next statement:

```
' X is passed twice; this will lead to alias problems
' in the Test procedure:
CALL Test(X, X, Y)
```

Example The following example illustrates how variable aliases can occur. Here the variable `A` is shared between the module-level code and the **SUB** procedure with the `DIM SHARED` statement. However, `A` is also passed by reference to the subprogram as an argument. Therefore, in the subprogram, `A` and `X` both refer to the same location in memory. Thus, when the subprogram modifies `X`, it is also modifying `A`, and vice versa.

```
DIM SHARED A
A = 4
CALL PrintHalf(A)
END

SUB PrintHalf (X) STATIC
  PRINT "Half of"; X; "plus half of"; A; "equals";
  X = X / 2       ' X and A now equal 2.
  A = A / 2       ' X and A now equal 1.
  PRINT A + X
END SUB
```

Output

```
Half of 4 plus half of 4 equals 2
```

Automatic and Static Variables

When the **STATIC** attribute appears on a procedure-definition line, it means that local variables within the procedure are "static"; that is, their values are preserved between calls to the procedure.

Leaving off the **STATIC** attribute makes local variables within the procedure "automatic" by default; that is, you get a fresh set of local variables each time the procedure is called.

You can override the effect of leaving off the **STATIC** attribute by using the **STATIC** statement within the procedure, thus making some variables automatic and others static (see the next section for more information).

Note The **SHARED** statement also overrides the default for variables in a procedure (local static or local automatic), since any variable appearing in a **SHARED** statement is known at the module level and thus is not local to the procedure.

Preserving Values of Local Variables with STATIC

Sometimes you may want to make some local variables in a procedure static while keeping the rest automatic. List those variables in a **STATIC** statement within the procedure.

Also, putting a variable name in a **STATIC** statement is a way of making absolutely sure that the variable is local, since a **STATIC** statement overrides the effect of a module-level **SHARED** statement.

Note

If you give a variable its type in an **AS** *type* clause, then the **AS** *type* clause must appear along with the variable's name in the **STATIC** and **DIM** statements.

A **STATIC** statement can appear only within a procedure. An array name in a **STATIC** statement must be followed by a set of empty parentheses. Also, you must dimension any array that appears in a **STATIC** statement before using the array, as shown in the next example:

```
SUB SubProg2
  STATIC Array() AS INTEGER
  DIM Array(-5 TO 5, 1 TO 25) AS INTEGER
  .
  .
  .
END SUB
```

Example

The following example shows how a **STATIC** statement preserves the value of the string variable `Y$` throughout successive calls to `TestSub`:

```
DECLARE SUB TestSub ()
FOR I% = 1 TO 5
  TestSub        ' Call TestSub five times.
NEXT I%
END

SUB TestSub              ' Note: no STATIC attribute.
 ' Both X$ and Y$ are local variables in TestSub (that is,
 ' their values are not shared with the module-level code).
 ' However since X$ is an automatic variable, it is
 ' reinitialized to a null string every time TestSub is
 ' called. In contrast, Y$ is static, so it retains the
 ' value it had from the last call:
  STATIC Y$
  X$ = X$ + "*"
  Y$ = Y$ + "*"
  PRINT X$, Y$
END SUB
```

Output

```
*          *
*          **
*          ***
*          ****
*          *****
```

Recursive Procedures

Procedures in BASIC can be recursive. A recursive procedure is one that can call itself or call other procedures that in turn call the first procedure.

The Factorial Function

A good way to illustrate recursive procedures is to consider the factorial function from mathematics. One way to define n! ("n factorial") is with the following formula:

$$n! = n * (n-1) * (n-2) * ... * 2 * 1$$

For example, 5 factorial is evaluated as follows:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Note

Do not confuse the mathematical factorial symbol (!) used in this discussion with the single-precision type-declaration suffix used by BASIC.

Factorials lend themselves to a recursive definition as well:

$$n! = n * (n-1)!$$

This leads to the following progression:

$$5! = 5 * 4!$$
$$4! = 4 * 3!$$
$$3! = 3 * 2!$$
$$2! = 2 * 1!$$
$$1! = 1 * 0!$$

Recursion must always have a terminating condition. With factorials, this terminating condition occurs when 0! is evaluated—by definition, 0! is equal to 1.

Note

Although a recursive procedure can have static variables by default (as in the next example), it is often preferable to let automatic variables be the default instead. In this way, recursive calls will not overwrite variable values from a preceding call.

Example The following example uses a recursive **FUNCTION** procedure to calculate factorials:

```
DECLARE FUNCTION Factorial# (N%)
DO
   INPUT "Enter number from 0 - 20 (or -1 to end): ", Num%
   IF Num% >= 0 AND Num% <= 20 THEN
      PRINT Num%; Factorial#(Num%)
   END IF
LOOP WHILE Num% >= 0
END

FUNCTION Factorial# (N%) STATIC

   IF N% > 0 THEN       ' Call Factorial# again
                        ' if N is greater than zero.
      Factorial# = N% * Factorial#(N% - 1)

   ELSE                 ' Reached the end of recursive calls
                        ' (N% = 0), so "climb back up the ladder."
      Factorial# = 1
   END IF
END FUNCTION
```

Adjusting the Size of the Stack

Recursion can eat up a lot of memory, since each set of automatic variables in a **SUB** or **FUNCTION** procedure is saved on the stack. (Saving variables this way allows a procedure to continue with the correct variable values after control returns from a recursive call.)

If you have a recursive procedure with many automatic variables, or a deeply nested recursive procedure, you may need to adjust the size of the stack before starting the procedure. Otherwise, you may get an `Out of stack space` error message.

To make this adjustment you use the **FRE** and **STACK** functions, plus the **STACK** statement as explained in the following.

Before actually adjusting the size of the stack, there are several facts that need to be determined. First, you must estimate the amount of memory your recursive procedure needs. Do this by following these steps:

1. Make a test module consisting of the **DECLARE** statement for the procedure, a single call to the procedure (using **CALL**), and the procedure itself.
2. Add a `FRE(-2)` function (which returns the total unused stack space) just before you call the recursive procedure. Add a second `FRE(-2)` function right at the end of the recursive procedure. Save the returned values in two long integers.
3. Run the test module. The difference in values is the amount of stack space (in bytes) used by one call to the procedure.

4. Estimate the maximum number of times the procedure is likely to be invoked, then multiply this value by the stack space consumed by one call to the procedure. The result is the amount of memory your recursive procedure needs.

Once you know how many bytes of stack space the procedure needs, you then determine the currently allocated size of the stack. This is 3K for DOS and 3.5K for OS/2 unless you have previously changed it with the **STACK** statement. Assuming that you are running under DOS and using the default stack size, the following code adjusts the size of the stack (if space is available):

```
' Initialize a variable that contains the currently allocated stack size.
CurrentSize = 3072
' Initialize a variable with the calculated recursion stack space
' requirements as explained above.
RecursiveBytes = 6000
' Find out how many bytes are used up on the stack right now.
BytesOnStack = CurrentSize - FRE(-2)
' Calculate the total required stack space.
RequiredSpace = RecursiveBytes + BytesOnStack
' Request the space if there's room.
IF RequiredSpace <= STACK THEN
   STACK RequiredSpace
   ELSE GOTO ReportError
END IF
```

Notice that in the preceding example, the **STACK** statement and **STACK** function were used. The **STACK** function returns the maximum space that can be allocated. The **STACK** statement allocates the space. See the *BASIC Language Reference* for further information.

Transferring Control to Another Program with CHAIN

Unlike procedure calls, which occur within the same program, the **CHAIN** statement simply starts a new program. When a program chains to another program, the following sequence occurs:

1. The first program stops running.
2. The second program is loaded into memory.
3. The second program starts running.

The advantage of using **CHAIN** is that it enables you to split a program with large memory requirements into several smaller programs.

The **COMMON** statement allows you to pass variables from one program to another program in a chain. A prevalent programming practice is to put these **COMMON** statements in an include file, and then use the **$INCLUDE** metacommand at the beginning of each program in the chain.

Note

Don't use a **COMMON** */blockname/ variablelist* statement (a "named **COMMON** block") to pass variables to a chained program, since variables listed in named **COMMON** blocks are not preserved when chaining. Use a blank **COMMON** block (**COMMON** *variablelist*) instead.

Example

This example, which shows a chain connecting three separate programs, uses an include file to declare variables passed in common among the programs:

```
' ============ CONTENTS OF INCLUDE FILE COMMONS.BI ========
DIM Values(10)
COMMON Values(), NumValues

' ======================== MAIN.BAS ========================
'
' Read in the contents of the COMMONS.BI file:
' $INCLUDE: 'COMMONS.BI'

   ' Input the data:
   INPUT "Enter number of data values (<=10): ", NumValues
   FOR I = 1 TO NumValues
      Prompt$ = "Value ("+LTRIM$(STR$(I))+")? "
      PRINT Prompt$;
      INPUT "", Values(I)
   NEXT I

   ' Have the user specify the calculation to do:
   INPUT "Calculation (1=st. dev., 2=mean)? ", Choice

   ' Now, chain to the correct program:
   SELECT CASE Choice
      CASE 1:           ' Standard deviation
    CHAIN "STDEV"

      CASE 2:           ' Mean
    CHAIN "MEAN"
   END SELECT
END
```

```
' ======================== STDEV.BAS ========================
' Calculates the standard deviation of a set of data
' ===========================================================
'
' $INCLUDE: 'COMMONS.BI'

   Sum   = 0    ' Normal sum
   SumSq = 0    ' Sum of values squared

   FOR I = 1 TO NumValues
      Sum   = Sum   + Values(I)
      SumSq = SumSq + Values(I) ^ 2
   NEXT I

   Stdev = SQR(SumSq / NumValues - (Sum / NumValues) ^ 2)
   PRINT "The Standard Deviation of the samples is: " Stdev
END

' ========================= MEAN.BAS ========================
' Calculates the mean (average) of a set of data
' ===========================================================
'
' $INCLUDE: 'COMMONS.BI'

   Sum = 0

   FOR I = 1 TO NumValues
      Sum = Sum + Values(I)
   NEXT

   Mean = Sum / NumValues
   PRINT "The mean of the samples is: " Mean
END
```

Sample Application: Recursive Directory Search (WHEREIS.BAS)

The following program uses a recursive **SUB** procedure, `ScanDir`, to scan a disk for the filename entered by the user. Each time this program finds the given file, it prints the complete directory path to the file.

Statements Used

This program demonstrates the following statements discussed in this chapter:

- **DECLARE**
- **FUNCTION...END FUNCTION**
- **STATIC**
- **SUB...END SUB**

Program Listing

```
DEFINT A-Z

' Declare symbolic constants used in program:
CONST EOFTYPE = 0, FILETYPE = 1, DIRTYPE = 2, ROOT = "TWH"

DECLARE SUB ScanDir (PathSpec$, Level, FileSpec$, Row)

DECLARE FUNCTION MakeFileName$ (Num)
DECLARE FUNCTION GetEntry$ (FileNum, EntryType)
CLS
INPUT "File to look for"; FileSpec$
PRINT
PRINT "Enter the directory where the search should start"
PRINT "(optional drive + directories). Press <ENTER> to "
PRINT "begin search in root directory of current drive."
PRINT
INPUT "Starting directory"; PathSpec$
CLS

RightCh$ = RIGHT$(PathSpec$, 1)

IF PathSpec$ = "" OR RightCh$ = ":" OR RightCh$ <> "\" THEN
   PathSpec$ = PathSpec$ + "\"
END IF
```

```
FileSpec$ = UCASE$(FileSpec$)
PathSpec$ = UCASE$(PathSpec$)
Level = 1
Row = 3

' Make the top level call (level 1) to begin the search:
ScanDir PathSpec$, Level, FileSpec$, Row

KILL ROOT + ".*"          ' Delete all temporary files created
                          ' by the program.

LOCATE Row + 1, 1: PRINT "Search complete."
END

' ====================== GetEntry =======================
'    This procedure processes entry lines in a DIR listing
'    saved to a file.

'    This procedure returns the following values:

'  GetEntry$     A valid file or directory name
'  EntryType     If equal to 1, then GetEntry$
'                is a file.
'                If equal to 2, then GetEntry$
'                is a directory.
' ======================================================
'
FUNCTION GetEntry$ (FileNum, EntryType) STATIC

   ' Loop until a valid entry or end-of-file (EOF) is read:
   DO UNTIL EOF(FileNum)
      LINE INPUT #FileNum, EntryLine$
      IF EntryLine$ <> "" THEN

         ' Get first character from the line for test:
       TestCh$ = LEFT$(EntryLine$, 1)
       IF TestCh$ <> " " AND TestCh$ <> "." THEN EXIT DO
      END IF
LOOP

' Entry or EOF found, decide which:
IF EOF(FileNum) THEN          ' EOF, so return EOFTYPE
   EntryType = EOFTYPE        ' in EntryType.
   GetEntry$ = ""

ELSE                          ' Not EOF, so it must be a
                              ' file or a directory.

   ' Build and return the entry name:
   EntryName$ = RTRIM$(LEFT$(EntryLine$, 8))
```

```
   ' Test for extension and add to name if there is one:
   EntryExt$ = RTRIM$(MID$(EntryLine$, 10, 3))
   IF EntryExt$ <> "" THEN
      GetEntry$ = EntryName$ + "." + EntryExt$
   ELSE
GetEntry$ = EntryName$
   END IF

   ' Determine the entry type, and return that value
   ' to the point where GetEntry$ was called:
   IF MID$(EntryLine$, 15, 3) = "DIR" THEN
EntryType = DIRTYPE              ' Directory
   ELSE
EntryType = FILETYPE             ' File
      END IF

   END IF

END FUNCTION

' ====================== MakeFileName$ ======================
'    This procedure makes a filename from a root string
'    ("TWH," defined as a symbolic constant at the module
'    level) and a number passed to it as an argument (Num).
' ===========================================================
'
FUNCTION MakeFileName$ (Num) STATIC

   MakeFileName$ = ROOT + "." + LTRIM$(STR$(Num))

END FUNCTION

' ======================== ScanDir ==========================
'   This procedure recursively scans a directory for the
'   filename entered by the user.
'
'    NOTE: The SUB header doesn't use the STATIC keyword
'          since this procedure needs a new set of variables
'          each time it is invoked.
' ===========================================================
'
SUB ScanDir (PathSpec$, Level, FileSpec$, Row)

   LOCATE 1, 1: PRINT "Now searching"; SPACE$(50);
   LOCATE 1, 15: PRINT PathSpec$;

   ' Make a file specification for the temporary file:
   TempSpec$ = MakeFileName$(Level)
```

```
    ' Get a directory listing of the current directory,
    ' and save it in the temporary file:
    SHELL "DIR " + PathSpec$ + " > " + TempSpec$

    ' Get the next available file number:
    FileNum = FREEFILE

    ' Open the DIR listing file and scan it:
    OPEN TempSpec$ FOR INPUT AS #FileNum
' Process the file, one line at a time:
    DO

       ' Input an entry from the DIR listing file:
       DirEntry$ = GetEntry$(FileNum, EntryType)

       ' If entry is a file:
       IF EntryType = FILETYPE THEN

    ' If the FileSpec$ string matches,
    ' print entry and exit this loop:
    IF DirEntry$ = FileSpec$ THEN
       LOCATE Row, 1: PRINT PathSpec$; DirEntry$;
       Row = Row + 1
       EntryType = EOFTYPE
    END IF

       ' If the entry is a directory, then make a recursive
       ' call to ScanDir with the new directory:
      ELSEIF EntryType = DIRTYPE THEN
    NewPath$ = PathSpec$ + DirEntry$ + "\"
    ScanDir NewPath$, Level + 1, FileSpec$, Row
    LOCATE 1, 1: PRINT "Now searching"; SPACE$(50);
    LOCATE 1, 15: PRINT PathSpec$;
      END IF

LOOP UNTIL EntryType = EOFTYPE

    ' Scan on this DIR listing file is finished, so close it:
    CLOSE FileNum
END SUB
```

Chapter 3
File and Device I/O

This chapter shows you how to use Microsoft BASIC input and output (I/O) functions and statements. These functions and statements permit your programs to access data stored in files and to communicate with devices attached to your system.

The chapter includes material on a variety of programming tasks related to retrieving, storing, and formatting information. The relationship between data files and physical devices such as screens and keyboards is also covered.

When you are finished with this chapter, you will know how to perform the following programming tasks:

- Print text on the screen.
- Get input from the keyboard for use in a program.
- Create data files on disk.
- Store records in data files.
- Read records from data files.
- Read or modify data in files that are not in ASCII format.
- Communicate with other computers through the serial port.

Note Creating and using ISAM files are discussed in Chapter 10, "Database Programming with ISAM."

Printing Text on the Screen

This section explains how to accomplish the following tasks:

- Display text on the screen using **PRINT**.
- Display formatted text on the screen using **PRINT USING**.
- Skip spaces in a row of printed text using **SPC**.
- Skip to a given column in a row of printed text using **TAB**.
- Change the number of rows or columns appearing on the screen using **WIDTH**.
- Open a text viewport using **VIEW PRINT**.

Note Output that appears on the screen is sometimes referred to as "standard output." You can redirect standard output by using the DOS command-line symbols > or >>, thus sending output that would have gone to the screen to a different output device (such as a printer) or to a disk file. (See your operating system documentation for more information on redirecting output.)

Screen Rows and Columns

To understand how text is printed on the screen, it helps to think of the screen as a grid of "rows" and "columns." The height of one row slightly exceeds the height of a line of printed output; the width of one column is just wider than the width of one character. A standard screen configuration in text mode (nongraphics) is 80 columns wide by 25 rows high. Figure 3.1 shows how each character printed on the screen occupies a unique cell in the grid, a cell that can be identified by pairing a row argument with a column argument.

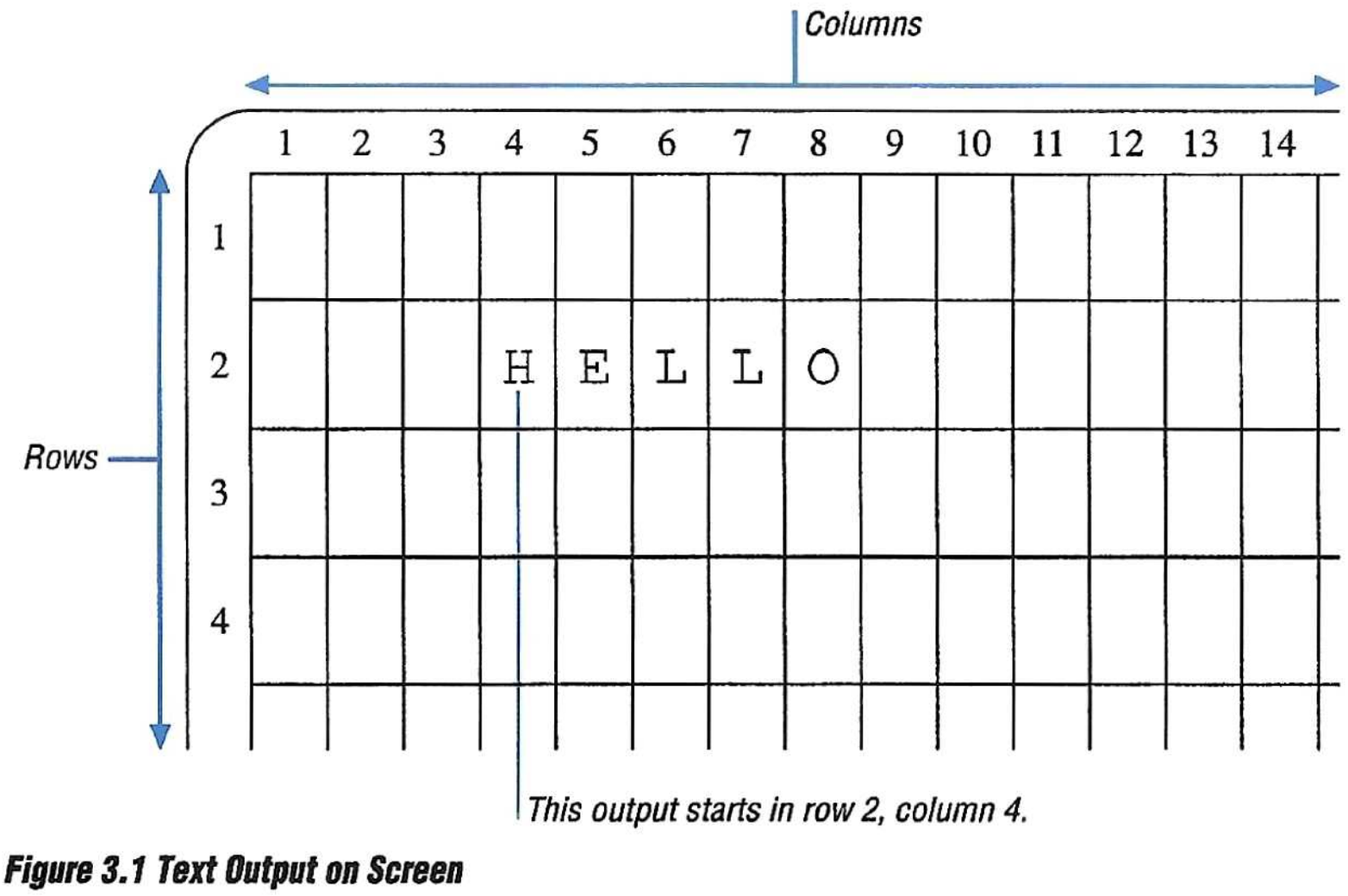


Figure 3.1 Text Output on Screen

The bottom row of the screen is not usually used for output, unless you use a **LOCATE** statement to display text there. (See the section "Controlling the Text Cursor" later in the chapter for more information on **LOCATE**.)

Displaying Text and Numbers with PRINT

By far the most commonly used statement for output to the screen is the **PRINT** statement. With **PRINT**, you can display numeric or string values, or a mixture of the two. In addition, **PRINT** with no arguments prints a blank line.

The following are some general comments about **PRINT**:

- **PRINT** always prints numbers with a trailing blank space. If the number is positive, the number is also preceded by a space; if the number is negative, the number is preceded by a minus sign (–).
- The **PRINT** statement can be used to print lists of expressions. Expressions in the list can be separated from other expressions by commas, semicolons, one or more blank spaces, or one or more tab characters. A comma causes **PRINT** to skip to the beginning of the next "print zone," or block of 14 columns, on the screen. A semicolon (or any combination of spaces and/or tabs) between two expressions prints the expressions on the screen next to each other, with no spaces in between (except for the built-in spaces for numbers).
- Ordinarily, **PRINT** ends each line of output with a new-line sequence (a carriage return and line feed). However, a comma or semicolon at the end of the list of expressions suppresses this; the next printed output from the program appears on the same line unless it is too long to fit on that line.
- **PRINT** wraps an output line that exceeds the width of the screen onto the next line. For example, if you try to print a line that is 100 characters long on an 80-column screen, the first 80 characters of the line show up on one row, followed by the next 20 characters on the next row. If the 100-character line didn't start at the left edge of the screen (for example, if it followed a **PRINT** statement ending in a comma or semicolon), then the line would print until it reached the 80th column of one row and continue in the first column of the next row.

Example The output from the following program shows some of the different ways you can use **PRINT**:

```
A = 2
B = -1
C = 3
X$ = "over"
Y$ = "there"

PRINT A, B, C
PRINT B, A, C
PRINT A; B; C
PRINT X$; Y$
PRINT X$, Y$;
PRINT A, B
PRINT
FOR I = 1 TO 8
   PRINT X$,
NEXT
```

Output

```
2              -1              3
-1               2               3
 2 -1  3
overthere
over           there 2         -1

over           over            over           over           over
over           over            over
```

Displaying Formatted Output with PRINT USING

The **PRINT USING** statement gives greater control than **PRINT** over the appearance of printed data, especially numeric data. Through the use of special characters embedded in a format string, **PRINT USING** allows you to specify information such as how many digits from a number (or how many characters from a string) are displayed, whether or not a plus sign (+) or a dollar sign ($) appears in front of a number, and so forth.

Example

The example that follows shows what can be done with **PRINT USING**. You can list more than one expression after the **PRINT USING** format string. As is the case with **PRINT**, the expressions in the list can be separated from one another by commas, semicolons, spaces, or tab characters.

```
X = 441.2318

PRINT USING "The number with 3 decimal places ###.###";X
PRINT USING "The number with a dollar sign $$##.##";X
PRINT USING "The number in exponential format #.###^^^^";X
PRINT USING "Numbers with plus signs +###  "; X; 99.9
```

Output

```
The number with 3 decimal places 441.232
The number with a dollar sign $441.23
The number in exponential format 0.441E+03
Numbers with plus signs +441  Numbers with plus signs +100
```

Consult online Help for more on **PRINT USING**.

Skipping Spaces and Advancing to a Specific Column

By using the SPC(*n*) statement in a **PRINT** statement, you can skip *n* spaces in a row of printed output, as shown in the next example:

```
PRINT "          1          2          3"
PRINT "1234567890123456789012345678 90"
PRINT "First Name"; SPC(10); "Last Name"
```

Output

```
1         2         3
12345678901234567890123456789O
First Name           Last Name
```

By using the **TAB(*n*)** statement in a **PRINT** statement, you can skip to the *n*th column (counting from the left side of the screen) in a row of printed output. In the following example, **TAB** produces the same output shown in the preceding example:

```
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
PRINT "First Name"; TAB(21); "Last Name"
```

Neither **SPC** nor **TAB** can be used by itself to position printed output on the screen; they can only appear in **PRINT** statements.

Changing the Number of Columns or Rows

You can control the maximum number of characters that appear in a single row of output by using the **WIDTH** *columns* statement. The **WIDTH** *columns* statement actually changes the size of characters that are printed on the screen, so that more or fewer characters can fit on a row. For example, `WIDTH 40` makes characters wider, so the maximum row length is 40 characters. `WIDTH 80` makes characters narrower, so the maximum row length is 80 characters. The numbers 40 and 80 are the only valid values for the *columns* argument.

On machines equipped with an Enhanced Graphics Adapter (EGA) or Video Graphics Array (VGA), the **WIDTH** statement can also control the number of rows that appear on the screen by using this syntax:

WIDTH [[*screenwidth%*]] [[,*screenheight%*]]

The value for *screenheight%* may be 25, 30, 43, 50, or 60, depending on the type of display adapter you use and the screen mode set in a preceding **SCREEN** statement.

Creating a Text Viewport

So far, the entire screen has been used for text output. However, with the **VIEW PRINT** statement, you can restrict printed output to a "text viewport," a horizontal slice of the screen. The syntax of the **VIEW PRINT** statement is:

VIEW PRINT [[*topline%* **TO** *bottomline%*]]

The values for *topline%* and *bottomline%* specify the locations where the viewport will begin and end, respectively.

A text viewport also gives you control over on-screen scrolling. Without a viewport, when printed output reaches the bottom of the screen, text or graphics output that was at the top of the screen scrolls off and is lost. However, after a **VIEW PRINT** statement, scrolling takes place only between the top and bottom lines of the viewport. This means you can label the displayed output at the top and/or bottom of the screen without having to worry that the labeling will scroll it off if too many lines of data appear. You can also use `CLS 2` to clear just the text viewport, leaving the contents of the rest of the screen intact. See the section "Defining a Graphics Viewport" in Chapter 5, "Graphics," to learn how to create a viewport for graphics output on the screen.

Example You can see the effects of a **VIEW PRINT** statement by examining the output from the next example:

```
CLS
LOCATE 3, 1
PRINT "This is above the text viewport; it doesn't scroll."

LOCATE 4, 1
PRINT STRING$(60, "_")        ' Print horizontal lines above
LOCATE 11, 1
PRINT STRING$(60, "_")        ' and below the text viewport.

PRINT "This is below the text viewport."

VIEW PRINT 5 TO 10            ' Text viewport extends from
                              ' lines 5 to 10.

FOR I = 1 TO 20               ' Print numbers and text in
   PRINT I; "a line of text" ' the viewport.
NEXT

DO: LOOP WHILE INKEY$ = ""    ' Wait for a key press.
CLS 2                         ' Clear just the viewport.
END
```

Output (Before User Presses Key)

```
This is above the text viewport; it doesn't scroll.

  16 a line of text
  17 a line of text
  18 a line of text
  19 a line of text
  20 a line of text


This is below the text viewport.
```

Output (After User Presses Key)

```
This is above the text viewport; it doesn't scroll.


This is below the text viewport.
```

Getting Input from the Keyboard

This section shows you how to use the following statements and functions to enable your BASIC programs to accept input entered from the keyboard:

- **INPUT**
- **LINE INPUT**
- **INPUT$**
- **INKEY$**

Note Input typed at the keyboard is often referred to as "standard input." You can use the DOS redirection symbol (<) to direct standard input to your program from a file or other input device instead of from the keyboard. (See your operating system documentation for more information on redirecting input.)

The INPUT Statement

The **INPUT** statement takes information typed by the user and stores it in a list of variables, as shown in the following example:

```
INPUT A%, B, C$
INPUT D$
PRINT A%, B, C$, D$
```

Output

```
? 6.6,45,a string
? "two, three"
 7             45            a string      two, three
```

Here are some general comments about **INPUT**:

- An **INPUT** statement by itself prompts the user with a question mark (?) followed by a blinking cursor.
- The **INPUT** statement is followed by one or more variable names. When there are two or more variables, they are separated by commas.
- The number of constants entered by the user after the **INPUT** prompt must be the same as the number of variables in the **INPUT** statement itself.
- The values the user enters must agree in type with the variables in the list following **INPUT**. In other words, enter a number if the variable is designated as having the type integer, long integer, single precision, or double precision. Enter a string if the variable is designated as having the type string.

- Since constants in an input list must be separated by commas, an input string constant containing one or more commas should be enclosed in double quotation marks. The double quotation marks ensure that the string is treated as a unit and not broken into two or more parts.

If the user breaks any of the last three rules, BASIC prints the error message `Redo from start.` This message reappears until the input agrees in number and type with the variable list.

If you want your input prompt to be more informative than a simple question mark, you can make a prompt appear, as in the following example:

```
INPUT "What is the correct time (hour, min)"; Hr$, Min$
```

This prints the following prompt:

```
What is the correct time (hour, min)?
```

Note the semicolon between the prompt and the input variables. This semicolon causes a question mark to appear as part of the prompt. Sometimes you may want to eliminate the question mark altogether; in this case, put a comma between the prompt and the variable list:

```
INPUT "Enter the time (hour, min): ", Hr$, Min$
```

This prints the following prompt:

```
Enter the time (hour, min):
```

The LINE INPUT Statement

If you want your program to accept lines of text with embedded commas, leading blanks, or trailing blanks, but you do not want to have to remind the user to enclose the input in double quotation marks, use the **LINE INPUT** statement. The **LINE INPUT** statement, as its name implies, accepts a line of input (terminated by pressing Enter) from the keyboard and stores it in a single string variable. Unlike **INPUT**, the **LINE INPUT** statement does not print a question mark by default to prompt for input; it does, however, allow you to display a prompt string.

Example The following example shows the difference between **INPUT** and **LINE INPUT**:

```
' Assign the input to three separate variables:
INPUT "Enter three values separated by commas: ", A$, B$, C$

' Assign the input to one variable (commas not treated
' as delimiters between input):
LINE INPUT "Enter the same three values: ", D$
PRINT "A$ = "; A$
PRINT "B$ = "; B$
PRINT "C$ = "; C$
PRINT "D$ = "; D$
```

Output

```
Enter 3 values separated by commas: by land, air, and sea
Enter the same three values: by land, air, and sea
A$ = by land
B$ = air
C$ = and sea
D$ = by land, air, and sea
```

With **INPUT** and **LINE INPUT**, input is terminated when the user presses Enter, which also advances the cursor to the next line. As the next example shows, a semicolon between the **INPUT** keyword and the prompt string keeps the cursor on the same line:

```
INPUT "First value: ", A
INPUT; "Second value: ", B
INPUT "    Third value: ", C
```

The following shows some sample input to the preceding program and the positions of the prompts:

```
First value: 5
Second value: 4      Third value: 3
```

The INPUT$ Function

INPUT and **LINE INPUT** wait for the user to press Enter before they store what is typed; that is, they read a line of input, then assign it to program variables. In contrast, the **INPUT$**(*number*) function doesn't wait for Enter to be pressed; it just reads a specified number of characters. For example, the following line in a program reads three characters typed by the user, then stores the three-character string in the variable `Test$`:

```
Test$ = INPUT$(3)
```

Unlike the **INPUT** statement, the **INPUT$** function does not prompt the user for data, nor does it echo input characters on the screen. Also, since **INPUT$** is a function, it cannot stand by itself as a complete statement. **INPUT$** must appear in an expression, as in the following:

```
INPUT X                 ' INPUT is a statement.

PRINT INPUT$(1)         ' INPUT$ is a function, so it must
Y$ = INPUT$(1)          ' appear in an expression.
```

The **INPUT$** function reads input from the keyboard as an unformatted stream of characters. Unlike **INPUT** or **LINE INPUT**, **INPUT$** accepts any key pressed, including control keys like Esc or Backspace. For example, pressing Enter five times assigns five carriage-return characters to the `Test$` variable in the next line:

```
Test$ = INPUT$(5)
```

The INKEY$ Function

The **INKEY$** function completes the list of BASIC's keyboard-input functions and statements. When BASIC encounters an expression containing the **INKEY$** function, it checks to see if the user has pressed a key since one of the following:

- The last time it found an expression with **INKEY$**
- The beginning of the program, if this is the first time **INKEY$** appears

If no key has been pressed since the last time the program checked, **INKEY$** returns a null string (""). If a key has been pressed, **INKEY$** returns the character corresponding to that key.

Example

The most important difference between **INKEY$** and the other statements and functions discussed in this section is that **INKEY$** lets your program continue doing other things while it checks for input. In contrast, **LINE INPUT**, **INPUT$**, and **INPUT** suspend program execution until there is input, as shown in this example:

```
PRINT "Press any key to start. Press any key to end."

' Don't do anything else until the user presses a key:
Begin$ = INPUT$(1)

I& = 1

' Print the numbers from one to one million.
' Check for a key press while the loop is executing:
DO
   PRINT I&
   I& = I& + 1

' Continue looping until the value of the variable I& is
' greater than one million or until a key is pressed:
LOOP UNTIL I& > 1000000 OR INKEY$ <> ""
```

Controlling the Text Cursor

When you display printed text on the screen, the text cursor marks the place on the screen where output from the program—or input typed by the user—will appear next. In the following example, after the **INPUT** statement displays its 12-character prompt, `"First name: "`, the cursor waits for input in row 1 at column 13:

```
' Clear the screen; start printing in row one, column one:
CLS
INPUT "First name: ", FirstName$
```

In the next example, the semicolon at the end of the second **PRINT** statement leaves the cursor in row 2 at column 27:

```
CLS
PRINT

' Twenty-six characters are in the next line:
PRINT "Press any key to continue.";
PRINT INPUT$(1)
```

The following sections show how to control the location of the text cursor, change its shape, and get information about its location.

Positioning the Cursor

The input and output statements and functions discussed so far do not allow much control over where output is displayed or where the cursor is located after the output is displayed. Input prompts or output always start in the far left column of the screen and descend one row at a time from top to bottom unless a semicolon is used in the **PRINT** or **INPUT** statements to suppress the carriage-return-and-line-feed sequence.

The **SPC** and **TAB** statements, discussed in the section "Skipping Spaces and Advancing to a Specific Column" later in this chapter give some control over the location of the cursor by allowing you to move it to any column within a given row.

The **LOCATE** statement extends this control one step further. The syntax for **LOCATE** is:

LOCATE [[*row%*]] [[, [[*column%*]] [[, [[*cursor%*]] [[, [[*start%*]] [[,*stop%*]]]]]]]]

Example

Using the **LOCATE** statement allows you to position the cursor in any row or column on the screen, as shown by the output in the next example:

```
CLS
FOR Row = 9 TO 1 STEP -2
    Column = 2 * Row
    LOCATE Row, Column
    PRINT "12345678901234567890";
NEXT
```

Output

```
12345678901234567890

    12345678901234567890

        12345678901234567890

            12345678901234567890

                12345678901234567890
```

Changing the Cursor's Shape

The optional *cursor%*, *start%*, and *stop%* arguments shown in the syntax for the **LOCATE** statement also allow you to change the shape of the cursor and make it visible or invisible. A value of 1 for *cursor%* makes the cursor visible, while a value of 0 makes the cursor invisible. The *start%* and *stop%* arguments control the height of the cursor, if it is on, by specifying the top and bottom "pixel" lines, respectively, for the cursor. (Any character on the screen is composed of lines of pixels, which are dots of light on the screen.) If a cursor spans the height of one row of text, then the line of pixels at the top of the cursor has the value 0, while the line of pixels at the bottom has a value of 7 or 13, depending whether your display adapter is monochrome (13) or color (7).

You can turn the cursor on and change its shape without specifying a new location for it. For example, the following statement keeps the cursor wherever it is at the completion of the next **PRINT** or **INPUT** statement, then makes it half a character high:

```
LOCATE , , 1, 2, 5  ' Row and column arguments both optional.
```

The following examples show different cursor shapes produced using different *start* and *stop* values on a color display. Each **LOCATE** statement shown in the left column is followed by the statement:

```
INPUT "PROMPT:", X$
```

Statement	Input prompt and cursor shape
`LOCATE , ,1,0,7`	`PROMPT:`▮
`LOCATE , ,1,0,3`	`PROMPT:`▪
`LOCATE , ,1,5,7`	`PROMPT:`▁
`LOCATE , ,1,6,2`	`PROMPT:`▪ ▪

In the preceding examples, note that making the *start%* argument bigger than the *stop%* argument results in a two-piece cursor.

Getting Information About the Cursor's Location

You can think of the functions **CSRLIN** and **POS(***numeric-expression***)** as the complements of the **LOCATE** statement: whereas **LOCATE** tells the cursor where to go, **CSRLIN** and **POS(***numeric-expression***)** tell your program where the cursor is. The **CSRLIN** function returns the current row and the **POS(***numeric-expression***)** function returns the current column of the cursor's position.

The argument *n* for **POS(***numeric-expression***)** is what is known as a "dummy" argument; that is, *numeric-expression* is a placeholder that can be any numeric expression. For example, `POS(0)` and `POS(1)` return the same value.

Example The following example uses the **POS(***numeric-expression***)** function to print 50 asterisks in rows of 13 asterisks:

```
FOR I% = 1 TO 50
   PRINT "*";                          ' Print an asterisk and keep
                                       ' the cursor on the same line.
   IF POS(1) > 13 THEN PRINT           ' If the cursor's position
                                       ' is past column 13, advance
                                       ' to the next line.
NEXT
```

Output

```
*************
*************
*************
**********
```

Working with Data Files

Data files are physical locations on your disk where information is permanently stored. The following tasks are greatly simplified by using data files in your BASIC programs:

- Creating, manipulating, and storing large amounts of data
- Accessing several sets of data with one program
- Using the same set of data in several different programs

The sections that follow introduce the concepts of records and fields and contrast different ways to access data files from BASIC. When you have completed those sections, you should know how to do the following:

- Create new data files
- Open existing files and read their contents
- Add new information to an existing data file
- Change the contents of an existing data file

How Data Files Are Organized

A data file is a collection of related blocks of information, or "records." Each record in a data file is further subdivided into "fields" or regularly recurring items of information within each record. If you compare a data file with a more old-fashioned way of storing information—for example, a folder containing application forms filled out by job applicants at a particular company—then a record is analogous to one application form in that folder. To carry the comparison one step further, a field is analogous to an item of information included on every application form, such as a Social Security number.

Note If you do not want to access a file using records but instead want to treat it as an unformatted sequence of bytes, then read the section "Binary File I/O" later in this chapter.

Sequential and Random-Access Files

The terms "sequential file" and "random-access file" refer to two different ways to store and access data on disk from your BASIC programs. A simplified way to think of these two kinds of files is with the following analogy: a sequential file is like a cassette tape, while a random-access file is like an LP record. To find a song on a cassette tape, you have to start at the beginning and fast-forward through the tape sequentially until you find the song you are looking for—there is no way to jump right to the song you want. This is similar to the way you have to find information in a sequential file: to get to the 500th record, you first have to read records 1 through 499.

In contrast, if you want to play a certain song on an LP, all you have to do is lift the tone arm of the record player and put the needle down right on the song: you can randomly access anything on the LP without having to play all the songs before the one you want. Similarly, you can call up any record in a random-access file just by specifying its number, greatly reducing access time.

Note Although there is no way to jump directly to a specific *record* in a sequential file, the **SEEK** statement lets you jump directly to a specific *byte* in the file. See the section "Binary File I/O" later in this chapter for more information on how to do this.

Opening a Data File

Before your program can read, modify, or add to a data file, it must first open the file. BASIC does this with the **OPEN** statement. The **OPEN** statement can be used to create a new file. The following list describes the various uses of the **OPEN** statement:

- Create a new data file and open it so records can be added to it. For example:

```
' No file named PRICE.DAT is in the current directory:
OPEN "PRICE.DAT" FOR OUTPUT AS #1
```

- Open an existing data file so new records overwrite any data already in the file. For example:

```
' A file named PRICE.DAT is already in the current
' directory; new records can be written to it, but all
' old records are lost:
OPEN "PRICE.DAT" FOR OUTPUT AS #1
```

- Open an existing data file so new records are added to the end of the file, preserving data already in the file. For example:

```
OPEN "PRICE.DAT" FOR APPEND AS #1
```

  The **APPEND** mode will also create a new file if a file with the given name does not already appear in the current directory.

- Open an existing data file so old records can be read from it. For example:

```
OPEN "PRICE.DAT" FOR INPUT AS #1
```

  See the section "Using Sequential Files" for more information about the **INPUT**, **OUTPUT**, and **APPEND** modes.

- Open an existing data file (or create a new one if a file with that name doesn't exist), then read or write fixed-length records to and from the file. For example:

```
OPEN "PRICE.DAT" FOR RANDOM AS #1
```

  See the section "Using Random-Access Files" for more information about this mode.

- Open an existing data file (or create a new one if a file with that name doesn't exist), then read data from the file or add new data to the file, starting at any byte position in the file. For example:

```
OPEN "PRICE.DAT" FOR BINARY AS #1
```

  See the section "Binary File I/O" for more information about this mode.

File Numbers in BASIC

The **OPEN** statement does more than just specify a mode for data I/O for a particular file (**OUTPUT**, **INPUT**, **APPEND**, **RANDOM**, or **BINARY**); it also associates a unique file number with that file. This file number, which can be any integer from 1 to 255, is then used by subsequent file I/O statements in the program as a shorthand way to refer to the file. As long as the file is open, this number remains associated with the file. When the file is closed, the file number is freed for use with another file. Your BASIC programs can open more than one file at a time.

The **FREEFILE** function can help you find an unused file number. This function returns the next available number that can be associated with a file in an **OPEN** statement. For example, **FREEFILE** might return the value 3 after the following **OPEN** statements:

```
OPEN "Test1.Dat" FOR RANDOM AS #1
OPEN "Test2.Dat" FOR RANDOM AS #2
FileNum = FREEFILE
OPEN "Test3.Dat" FOR RANDOM AS #FileNum
```

The **FREEFILE** function is particularly useful when you create your own library procedures that open files. With **FREEFILE**, you don't have to pass information about the number of open files to these procedures.

Filenames in BASIC

Filenames in **OPEN** statements can be any string expression, composed of any combination of the following characters:

- The letters a–z and A–Z
- The numbers 0–9
- The following special characters:

  ( ) @ # $ % ^ & ! - _ ' ~

The string expression can also contain an optional drive, as well as a complete or partial path specification. This means your BASIC program can work with data files on another drive or in a directory other than the one where the program is running. For example, the following **OPEN** statements are all valid:

```
OPEN "..\Grades.Qtr" FOR INPUT AS #1

OPEN "A:\SALARIES\1990.MAN" FOR INPUT AS #2

FileName$ = "TempFile"
OPEN FileName$ FOR OUTPUT AS #3

BaseName$ = "Invent"
OPEN BaseName$ + ".DAT" FOR OUTPUT AS #4
```

DOS also imposes its own restrictions on filenames: you can use no more than eight characters for the filename (everything to the left of an optional period) and no more than three characters for the extension (everything to the right of an optional period).

Long filenames in BASIC programs are truncated in the following fashion:

Filename in program	Resulting filename in DOS
`Prog@Data@File`	`PROG@DAT.A@F` The BASIC name is more than 11 characters long, so BASIC takes the first eight characters for the base name, inserts a period (.), and uses the next three characters as the extension. Everything else is discarded.
`Mail#.Version1`	`MAIL#.VER` The filename (`Mail#`) is shorter than eight characters, but the extension (`Version1`) is longer than three, so the extension is shortened to three characters.
`RELEASE_NoteS.BAK`	Gives the run-time error message `Bad file name`. The base name must be shorter than eight characters if you are going to include an explicit extension (.BAK in this case).

DOS is not case sensitive, so lowercase letters in filenames are converted to all uppercase (capital) letters. Therefore, you should not rely on the mixing of lowercase and uppercase to distinguish between files. For example, if you already had a file on the disk named INVESTOR.DAT, the following **OPEN** statement would overwrite that file, destroying any information already stored in it:

```
OPEN "Investor.Dat" FOR OUTPUT AS #1
```

Closing a Data File

Closing a data file has two important results: first, it writes any data currently in the file's buffer (a temporary holding area in memory) to the file; second, it frees the file number associated with that file for use by another **OPEN** statement.

Use the **CLOSE** statement following a program to close a file. For example, consider a file `PRICE.DAT` that is opened with this statement:

```
OPEN "PRICE.DAT" FOR OUTPUT AS #1
```

The statement `CLOSE #1` then ends output to PRICE.DAT. Next, PRICE.DAT is opened with the following:

```
OPEN "PRICE.DAT" FOR OUTPUT AS #2
```

Then the appropriate statement for ending output is `CLOSE #2`. A **CLOSE** statement with no file number arguments closes all open files.

A data file is also closed when either of the following occurs:

- The BASIC program performing I/O ends. (Program termination always closes all open data files.)
- The program performing I/O transfers control to another program with the **RUN** statement (or with the **CHAIN** statements if compiled with the /O option).

Using Sequential Files

This section discusses how records are organized in sequential data files and then shows how to read data from, or write data to, an open sequential file.

Records in Sequential Files

Sequential files are ASCII (text) files. This means you can use any word processor to view or modify a sequential file. Records are stored in sequential files as a single line of text, terminated by a carriage-return-and-line-feed (CR-LF) sequence. Each record is divided into fields, or repeated chunks of data that occur in the same order in every record. Figure 3.2 shows how three records might appear in a sequential file.

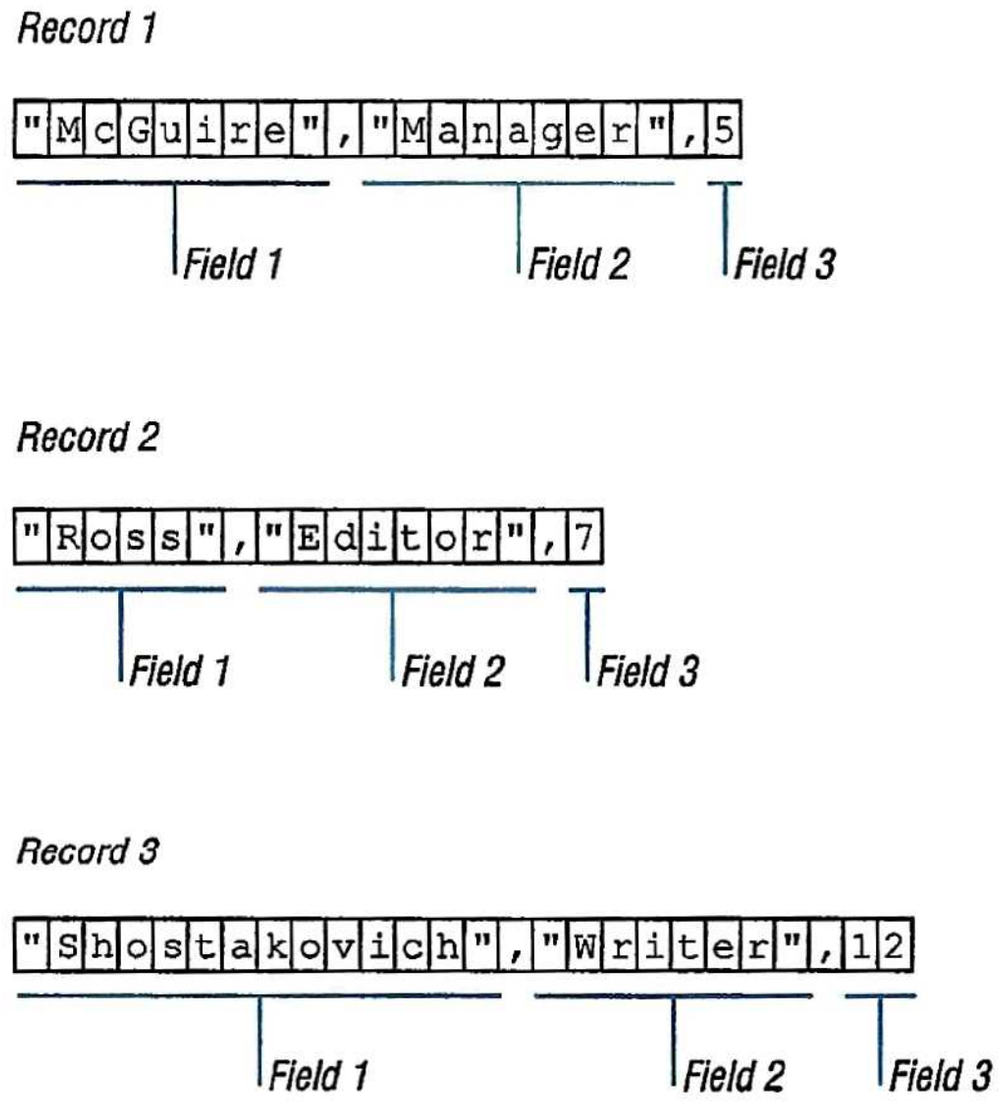


Figure 3.2 Records in Sequential Files

Note that each record in a sequential file can be a different length; moreover, fields can be different lengths in different records.

The kind of variable in which a field is stored determines where that field begins and ends. (See the following sections for examples of reading and storing fields from records.) For example, if your program reads a field into a string variable, then any of the following can signal the end of that field:

- Double quotation marks (") if the string begins with double quotation marks
- Comma (,) if the string does not begin with double quotation marks
- CR-LF if the field is at the end of the record

On the other hand, if your program reads a field into a numeric variable, then any of the following can signal the end of that field:

- Comma
- One or more spaces
- CR-LF

Putting Data in a New Sequential File

You can add data to a new sequential file after first opening it to receive records with an **OPEN** *filename* **FOR OUTPUT** statement. Use the **WRITE #** statement to write records to the file.

You can open sequential files for reading or for writing but not for both at the same time. If you are writing to a sequential file and want to read back the data you stored, you must first close the file, then reopen it for input.

Example The following short program creates a sequential file named `PRICE.DAT`, then adds data entered at the keyboard to the file. The **OPEN** statement in this program creates the file and readies it to receive records. The **WRITE #** statement then writes each record to the file. Note that the number used in the **WRITE #** statement is the same number given to the filename PRICE.DAT in the **OPEN** statement.

```
' Create a file named PRICE.DAT
' and open it to receive new data:
OPEN "PRICE.DAT" FOR OUTPUT AS #1

DO
   ' Continue putting new records in PRICE.DAT until the
   ' user presses Enter without entering a company name:
   INPUT "Company (press <ENTER> to quit): ", Company$
```

```
    IF Company$ <> "" THEN

       ' Enter the other fields of the record:
       INPUT "Style: ", Style$
       INPUT "Size: ", Size$
       INPUT "Color: ", Clr$
       INPUT "Quantity: ", Qty

       ' Put the new record in the file
       ' with the WRITE # statement:
       WRITE #1, Company$, Style$, Size$, Clr$, Qty
    END IF
LOOP UNTIL Company$ = ""

' Close PRICE.DAT (this ends output to the file):
CLOSE #1
END
```

Warning If, in the case of the preceding example, you already had a file named `PRICE.DAT` on the disk, the **OUTPUT** mode given in the **OPEN** statement would erase the existing contents of PRICE.DAT before writing any new data to it. If you want to add new data to the end of an existing file without erasing what is already in it, use the **APPEND** mode of **OPEN.** See the section "Adding Data to a Sequential File" later in this chapter for more information on this mode.

Reading Data from a Sequential File

You can read data from a sequential file after first opening it with the statement **OPEN** *filename* **FOR INPUT.** Use the **INPUT#** statement to read records from the file one field at a time. (See the section "Other Ways to Read Data from a Sequential File" later in this chapter for information on other file-input statements and functions you can use with a sequential file.)

Example The following program opens the `PRICE.DAT` data file created in the previous example and reads the records from the file, displaying the complete record on the screen if the quantity for the item is less than the input amount.

The `INPUT #1` statement reads one record at a time from PRICE.DAT, assigning the fields in the record to the variables `Company$`, `Style$`, `Size$`, `Clr$`, and `Qty`. Since this is a sequential file, the records are read in order from the first to the last entered.

The **EOF** (end-of-file) function tests whether the last record has been read by **INPUT#.** If the last record has been read, **EOF** returns the value –1 (true), and the loop for getting data ends; if the last record has not been read, **EOF** returns the value 0 (false), and the next record is read from the file.

```
OPEN "PRICE.DAT" FOR INPUT AS #1

INPUT "Display all items below what level"; Reorder

DO UNTIL EOF(1)
   INPUT #1, Company$, Style$, Size$, Clr$, Qty
   IF Qty < Reorder THEN
      PRINT  Company$, Style$, Size$, Clr$, Qty
   END IF
LOOP
CLOSE #1
END
```

Adding Data to a Sequential File

As mentioned earlier, if you have a sequential file on disk and want to add more data to the end of it, you cannot simply open the file in output mode and start writing data. As soon as you open a sequential file in output mode, you destroy its current contents. You must use the **APPEND** mode instead, as shown in the next example:

```
OPEN "PRICE.DAT" FOR APPEND AS #1
```

APPEND is always a safe alternative to **OUTPUT**, since **APPEND** creates a new file if one with the name specified doesn't already exist. For example, if a file named `PRICE.DAT` did not reside on disk, the example statement would make a new file with that name.

Other Ways to Write Data to a Sequential File

The preceding examples all use the **WRITE #** statement to write records to a sequential file. There is, however, another statement you can use to write sequential file records: **PRINT #.**

The best way to show the difference between these two data-storage statements is to examine the contents of a file created with both. The following short fragment opens a file named `TEST.DAT`, then places the same record in it twice, once with **WRITE #** and once with **PRINT #.** After running this program you can examine the contents of `TEST.DAT` with the DOS TYPE command.

```
OPEN "TEST.DAT" FOR OUTPUT AS #1
Nm$ = "Penn, Will"
Dept$ = "User Education"
Level = 4
Age = 25
WRITE #1, Nm$, Dept$, Level, Age
PRINT #1, Nm$, Dept$, Level, Age
CLOSE #1
```

Output

```
"Penn, Will","User Education",4,25
Penn, Will    User Education                4              25
```

The record stored with **WRITE #** has commas that explicitly separate each field of the record, as well as double quotation marks enclosing each string expression. On the other hand, **PRINT #** has written an image of the record to the file exactly as it would appear on screen with a simple **PRINT** statement. The commas in the **PRINT #** statement are interpreted as meaning "advance to the next print zone" (a new print zone occurs every 14 spaces, starting at the beginning of a line), and quotation marks are not placed around the string expressions.

At this point, you may be wondering what difference these output statements make, except in the appearance of the data within the file. The answer lies in what happens when your program reads the data back from the file with an **INPUT #** statement. In the following example, the program reads the record stored with **WRITE #** and prints the values of its fields without any problem:

```
OPEN "TEST.DAT" FOR INPUT AS #1
' Input the first record,
' and display the contents of each field:
INPUT #1, Nm$, Dept$, Level, Age
PRINT Nm$, Dept$, Level, Age


' Input the second record,
' and display the contents of each field:
INPUT #1, Nm$, Dept$, Level, Age
PRINT Nm$, Dept$, Level, Age

CLOSE #1
```

Output

```
Penn, Will     User Education                  4              25
```

However, when the program tries to input the next record stored with **PRINT #**, the attempt produces the error message `Input past end of file`. Without double quotation marks enclosing the first field, the **INPUT #** statement sees the comma between `Penn` and `Will` as a field delimiter, so it assigns only the last name `Penn` to the variable `Nm$`. **INPUT #** then reads the rest of the line into the variable `Dept$`. Since all of the record has now been read, there is nothing left to put in the variables `Level` and `Age`. The result is the error message `Input past end of file`.

If you are storing records that have string expressions and you want to read these records later with the **INPUT #** statement, follow one of these two rules of thumb:

- Use the **WRITE #** statement to store the records.
- If you want to use the **PRINT #** statement, remember it does not put commas in the record to separate fields, nor does it put double quotation marks around strings. You have to put these field separators in the **PRINT #** statement yourself.

Example For example, you can avoid the problems shown in the preceding example by using **PRINT #** with double quotation marks surrounding each field containing a string expression, as in the following example:

```
' 34 is ASCII value for double-quotation-mark character:

Q$ = CHR$(34)

' The next four statements all write the record to the
' file with double quotation marks around each string field:

PRINT #1, Q$ Nm$ Q$ Q$ Dept$ Q$ Level Age
PRINT #1, Q$ Nm$ Q$;Q$ Dept$ Q$;Level;Age
PRINT #1, Q$ Nm$ Q$,Q$ Dept$ Q$,Level,Age
WRITE #1, Nm$, Dept$, Level, Age
```

Output to File

```
"Penn, Will""User Education" 4  25
"Penn, Will""User Education" 4  25
"Penn, Will"  "User Education"      4              25
"Penn, Will","User Education",4,25
```

Other Ways to Read Data from a Sequential File

In the preceding sections, **INPUT #** is used to read a record (one line of data from a file), assigning different fields in the record to the variables listed after **INPUT #**. This section explores alternative ways to read data from sequential files, as records (**LINE INPUT #**) and as unformatted sequences of bytes (**INPUT$**).

The LINE INPUT # Statement

With the **LINE INPUT#** statement, your program can read a line of text exactly as it appears in a file without interpreting commas or double quotation marks as field delimiters. This is particularly useful in programs that work with ASCII text files.

The **LINE INPUT #** statement reads an entire line from a sequential file (up to a carriage-return-and-line-feed sequence) into a single string variable.

Examples The following short program reads each line from the file `CHAP1.TXT` and then echoes that line on the screen:

```
' Open CHAP1.TXT for sequential input:
OPEN "CHAP1.TXT" FOR INPUT AS #1

' Keep reading lines sequentially from the file until
' there are none left in the file:
DO UNTIL EOF(1)
```

```
    ' Read a line from the file and store it
    ' in the variable LineBuffer$:
    LINE INPUT #1, LineBuffer$

    ' Print the line on the screen:
    PRINT LineBuffer$
LOOP
```

The preceding program is easily modified to a file-copying utility that prints each line read from the specified input file to another file, instead of to the screen:

```
' Input names of input and output files:

INPUT "File to copy: ", FileName1$
IF FileName1$ = "" THEN END
INPUT "Name of new file: ", FileName2$
IF FileName2$ = "" THEN END

' Open first file for sequential input:
OPEN FileName1$ FOR INPUT AS #1

' Open second file for sequential output:
OPEN FileName2$ FOR OUTPUT AS #2

' Keep reading lines sequentially from first file
' until there are none left in the file:
DO UNTIL EOF(1)

    ' Read a line from first file and store it in the
    ' variable LineBuffer$:
    LINE INPUT #1, LineBuffer$

    ' Write LineBuffer$ to the second file:
    PRINT #2, LineBuffer$

LOOP
```

The INPUT$ Function

Another way to read data from sequential files (and, in fact, from any file) is to use the **INPUT$** function. Whereas **INPUT #** and **LINE INPUT #** read one line at a time from a sequential file, **INPUT$** reads a specified number of characters from a file, as shown in the following examples:

Statement	Action
`X$ = INPUT$(100, #1)`	Reads 100 characters from file number 1 and assigns all of them to the string variable `X$`.
`Test$ = INPUT$(1, #2)`	Reads one character from file number 2 and assigns it to the string variable `Test$`.

The **INPUT$** function without a file number always reads input from standard input (usually the keyboard).

The **INPUT$** function does what is known as "binary input"; that is, it reads a file as an unformatted stream of characters. For example, it does not see a carriage-return-and-line-feed sequence as signalling the end of an input operation. Therefore, **INPUT$** is the best choice when you want your program to read every single character from a file or when you want it to read a binary, or non-ASCII, file.

Example The following program copies the named binary file to the screen, printing only alphanumeric and punctuation characters in the ASCII range 32 to 126, as well as tabs, carriage returns, and line feeds:

```
' 9 is ASCII value for horizontal tab, 10 is ASCII value
' for line feed, and 13 is ASCII value for carriage return:
CONST LINEFEED = 10, CARRETURN = 13, TABCHAR = 9

INPUT "Print which file: ", FileName$
IF FileName$ = "" THEN END

OPEN FileName$ FOR INPUT AS #1

DO UNTIL EOF(1)
   Character$ = INPUT$(1, #1)
   CharVal = ASC(Character$)
   SELECT CASE CharVal
      CASE 32 TO 126
    PRINT Character$;
      CASE TABCHAR, CARRETURN
    PRINT Character$;
      CASE LINEFEED
         IF OldCharVal <> CARRETURN THEN PRINT Character$;
      CASE ELSE
    ' This is not one of the characters this program
    ' is interested in, so don't print anything.
    END SELECT
    OldCharVal = CharVal
LOOP
```

Using Random-Access Files

This section discusses how records are organized in random-access data files, then shows you how to read data from, and write data to, a file opened for random access.

Records in Random-Access Files

Random-access records are stored quite differently from sequential records. Each random-access record is defined with a fixed length, as is each field within the record. These fixed lengths determine where a record or field begins and ends, as there are no commas separating fields, and no carriage-return-and-line-feed sequences between records. Figure 3.3 shows how three records might appear in a random-access file.

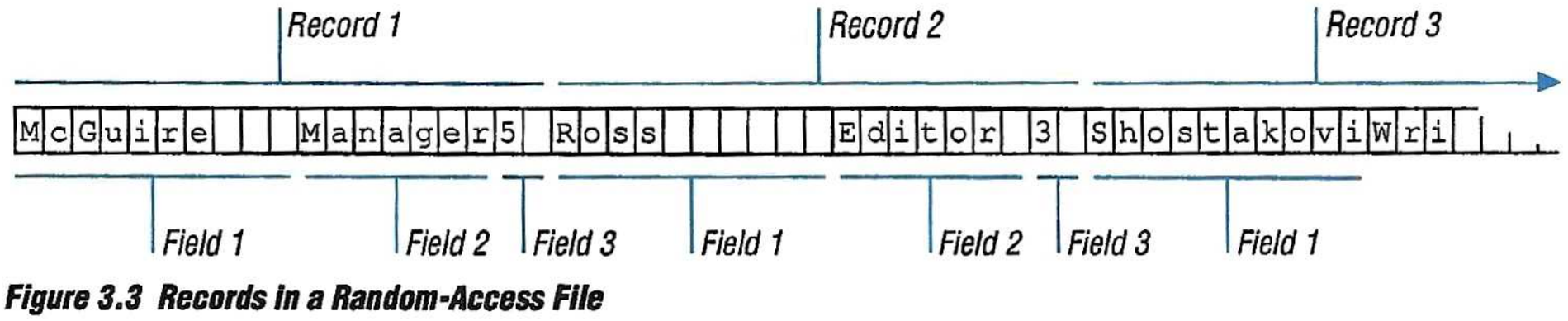


Figure 3.3 Records in a Random-Access File

If you are storing records containing numbers, using random-access files saves disk space when compared with using sequential files. This is because sequential files save numbers as a sequence of ASCII characters representing each digit, whereas random-access files save numbers in binary format.

For example, the number 17,000 is represented in a sequential file using 5 bytes, one for each digit. However, if 17,000 is stored in an integer field of a random-access record, it takes only 2 bytes of disk space.

Integers in random-access files take 2 bytes, long integers and single-precision numbers take 4 bytes, and double-precision numbers and currency data types take 8 bytes.

Adding Data to a Random-Access File

To write a program that adds data to a random-access file, follow these steps:

1. Define the fields of each record.
2. Open the file in random-access mode and specify the length of each record.
3. Get input for a new record and store the record in the file.

Each of these steps is now considerably easier than it was in BASICA, as you can see from the examples that follow.

Defining Records

You can define your own record with a **TYPE...END TYPE** statement, which allows you to create a composite data type that mixes string and numeric elements. This is a big advantage over the earlier method of setting up records with a **FIELD** statement, which required that each field be defined as a string. By defining a record with **TYPE...END TYPE**, you eliminate the

need to use the functions that convert numeric data to strings (**MK*type*$**, **MK$**, **MKSMBF$**, and **MKDMBF$**) and strings to numeric data (**CV*type***, **CVSMBF**, and **CVDMBF**).

The following examples contrast these two methods of defining records.

- Record defined with **TYPE...END TYPE**:

```
' Define the RecordType structure:
TYPE RecordType
   Name AS STRING * 30
   Age AS INTEGER
   Salary AS SINGLE
END TYPE

' Declare the variable RecordVar
' as having the type RecordType:
DIM RecordVar AS RecordType
.
.
.
```

- Record defined with **FIELD**:

```
' Define the lengths of the fields
' in the temporary storage buffer:
FIELD #1,30 AS Name$,2 AS Age$,4 AS Salary$
.
.
.
```

Opening the File and Specifying Record Length

Since the length of a random-access record is fixed, you should let your program know how long you want each record to be; otherwise, record length defaults to 128 bytes.

To specify record length, use the **LEN =** clause in the **OPEN** statement. The next two fragments, which continue the contrasting examples started in the preceding section, show how to use **LEN =**.

- Specify the record length for a record that is defined with the statement **TYPE...END TYPE**:

```
.
.
.
' Open the random-access file and specify the length
' of one record as being equal to the length of the
' RecordVar variable:
OPEN "EMPLOYEE.DAT" FOR RANDOM AS #1 LEN = LEN(RecordVar)
.
.
.
```

- Specify record length for a record defined with **FIELD**:

```
.
.
.
' Open the random-access file and specify the length
' of a record:
OPEN "EMPLOYEE.DAT" FOR RANDOM AS #1 LEN = 36
.
.
.
```

As you can see, when you use **FIELD**, you have to add the lengths of each field yourself (`Name$` is 30 bytes, `Age$` is 2 bytes, `Salary$` is 4 bytes, so the record is 30+2+4 or 36 bytes). With **TYPE...END TYPE**, you no longer have to do these calculations. Instead, just use the **LEN** function to calculate the length of the variable you have created to hold your records (`RecordVar`, in this case).

Entering Data and Storing the Record

You can enter data directly into the elements of a user-defined record without having to worry about left or right justification of input within a field with **LSET** or **RSET**. Compare the following two fragments, which continue the examples started in the preceding section, to see the amount of code this approach saves you.

- Enter data for a random-access record and storing the record using **TYPE...END TYPE**:

```
.
.
.

' Enter the data:
INPUT "Name"; RecordVar.Name
INPUT "Age"; RecordVar.Age
INPUT "Salary"; RecordVar.Salary
' Store the record:
PUT #1, , RecordVar

.
.
.
```

- Enter data for a random-access record and store the record using **FIELD:**

```
.
.
.

' Enter the data:
INPUT "Name"; Nm$
INPUT "Age"; AgeVal%
INPUT "Salary"; SalaryVal!

' Left justify the data in the storage-buffer fields,
' using the MKI$ and MKS$ functions to convert numbers
' to file strings:
LSET Name$ = Nm$
LSET Age$ = MKI$(AgeVal%)
LSET Salary$ = MKS$(SalaryVal!)

' Store the record:
PUT #1

.
.
.
```

Putting It All Together

The following program puts together all the steps outlined in the preceding section—defining fields, specifying record length, entering data, and storing the input data—to open a random-access data file named `STOCK.DAT` and add records to it:

```
DEFINT A-Z

' Define structure of a single record in the random-access
' file. Each record will consist of four fixed-length fields
' ("PartNumber", "Description", "UnitPrice", "Quantity"):
TYPE StockItem
   PartNumber AS STRING *  6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE
' Declare a variable (StockRecord) using the above type:
DIM StockRecord AS StockItem

' Open the random-access file, specifying the length of one
' record as the length of the StockRecord variable:
OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=LEN(StockRecord)
```

```
' Use LOF() to calculate the number of records already in
' the file, so new records will be added after them:
RecordNumber = LOF(1) \ LEN(StockRecord)

' Now, add new records:
DO

   ' Input data for a stock record from keyboard and store
   ' in the different elements of the StockRecord variable:
   INPUT "Part Number? ", StockRecord.PartNumber
   INPUT "Description? ", StockRecord.Description
   INPUT "Unit Price ? ", StockRecord.UnitPrice
   INPUT "Quantity   ? ", StockRecord.Quantity

   RecordNumber = RecordNumber + 1

   ' Write data in StockRecord to a new record in the file:
   PUT #1, RecordNumber, StockRecord

   ' Check to see if more data are to be read:
   INPUT "More (Y/N)? ", Resp$
LOOP UNTIL UCASE$(Resp$) = "N"

' All done; close the file and end:
CLOSE  #1
END
```

If the `STOCK.DAT` file already existed, this program would add more records to the file without overwriting any that were already in the file. The following key statement makes this work:

```
RecordNumber = LOF(1) \ LEN(StockRecord)
```

Here is what happens:

1. The `LOF(1)` function calculates the total number of bytes in the file `STOCK.DAT`. If STOCK.DAT is new or has no records in it, `LOF(1)` returns the value 0.
2. The `LEN(StockRecord)` function calculates the number of bytes in one record. (`StockRecord` is defined as having the same structure as the user-defined type `StockItem`.)
3. Therefore, the number of records is equal to the total bytes in the file divided by the bytes in one record. This is another advantage of having a fixed-length record. Since each record is the same size, you can always use a formula to calculate the number of records in the file. Obviously, this would not work with a sequential file, since sequential records can have different lengths.

Reading Data Sequentially

Using the technique outlined in the preceding section for calculating the number of records in a random-access file, you can write a program that reads all the records in that file.

Example The following program reads records sequentially (from the first record stored to the last) from the `STOCK.DAT` file created in the previous section:

```
' Define a record structure (type) for random-access
' file records:
TYPE StockItem
   PartNumber AS STRING *  6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

' Declare a variable (StockRecord) using the above type:
DIM StockRecord AS StockItem

' Open the random-access file:
OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN = LEN(StockRecord)

' Calculate number of records in the file:
NumberOfRecords = LOF(1) \ LEN(StockRecord)

' Read the records and write the data to the screen:
FOR RecordNumber = 1 TO NumberOfRecords

   ' Read the data from a new record in the file:
   GET #1, RecordNumber, StockRecord

   ' Print the data to the screen:
   PRINT "Part Number: ", StockRecord.PartNumber
   PRINT "Description: ", StockRecord.Description
   PRINT "Unit Price : ", StockRecord.UnitPrice
   PRINT "Quantity   : ", StockRecord.Quantity

NEXT

' All done; close the file and end:
CLOSE #1
END
```

It is not necessary to close STOCK.DAT before reading from it. Opening a file for random access lets you write to or read from the file with a single **OPEN** statement.

Using Record Numbers to Retrieve Records

You can read any record from a random-access file by specifying the record's number in a **GET** statement. You can write to any record in a random-access file by specifying the record's number in a **PUT** statement. This is one of the major advantages that random-access files have over sequential files, since sequential files do not permit direct access to a specific record.

The sample application program, `INDEX.BAS`, listed in the section "Indexing a Random-Access File" later in this chapter shows a technique that quickly finds a particular record by searching an index of record numbers.

Example The following fragment shows how to use **GET** with a record number:

```
DEFINT A-Z    ' Default variable type is integer.
CONST FALSE = 0, TRUE = NOT FALSE

TYPE StockItem
   PartNumber AS STRING *  6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

DIM StockRecord AS StockItem

OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN=LEN(StockRecord)

NumberOfRecords = LOF(1) \ LEN(StockRecord)
GetMoreRecords = TRUE

DO
   PRINT "Enter record number for part you want to see ";
   PRINT "(0 to end): ";
   INPUT "", RecordNumber

   IF RecordNumber>0 AND RecordNumber<NumberOfRecords THEN

' Get the record whose number was entered and store
' it in StockRecord:
GET #1, RecordNumber, StockRecord

' Display the record:
.
.
.
```

```
    ELSEIF RecordNumber = 0 THEN
       GetMoreRecords = FALSE
    ELSE
       PRINT "Input value out of range."
    END IF
LOOP WHILE GetMoreRecords
END
```

Binary File I/O

Binary access is a third way—in addition to random access and sequential access—to read or write a file's data. Use the following statement to open a file for binary I/O:

OPEN *file$* **FOR BINARY AS** *#filenumber%*

Binary access is a way to get at the raw bytes of any file, not just an ASCII file. This makes it very useful for reading or modifying files saved in a non-ASCII format, such as Microsoft Word files or executable files.

Files opened for binary access are treated as an unformatted sequence of bytes. Although you can read or write a record (a variable declared as having a user-defined type) to a file opened in the binary mode, the file itself does not have to be organized into fixed-length records. In fact, binary I/O does not have to deal with records at all, unless you consider each byte in a file as a separate record.

Comparing Binary Access and Random Access

The **BINARY** mode is similar to the **RANDOM** mode in that you can both read from and write to a file after a single **OPEN** statement. (Binary thus differs from sequential access, where you must first close a file and then reopen it if you want to switch between reading and writing.) Also, like random access, binary access lets you move backward and forward within an open file. Binary access even supports the same statements used for reading and writing random-access files using this syntax:

{GET | PUT} [[#]]*filenumber%* [[, [[*position%*]]]] [[, [[*variable*]]]]

Here, *variable* can have any type, including a variable-length string or a user-defined type, and *position%* points to the place in the file where the next **GET** or **PUT** operation will take place. (The *position%* value is relative to the beginning of the file; that is, the first byte in the file has position one, the second byte has position two, and so on.) If you leave off the *position%* argument, successive **GET** and **PUT** operations move the file pointer sequentially through the file from the first byte to the last.

The **GET** statement reads a number of bytes from the file equal to the length *variable*. Similarly, the **PUT** statement writes a number of bytes to the file equal to the length *variable*.

For example, if *variable* has integer type, then **GET** reads 2 bytes into *variable*; if *variable* has single-precision type, **GET** reads 4 bytes. Therefore, if you don't specify a *position* argument in a **GET** or **PUT** statement, the file pointer is moved a distance equal to the length *variable*.

The **GET** statement and **INPUT$** function are the only ways to read data from a file opened in binary mode. The **PUT** statement is the only way to write data to a file opened in binary mode.

Binary access, unlike random access, enables you to move to any byte position in a file and then read or write any number of bytes you want. In contrast, random access can only move to a record boundary and read a fixed number of bytes (the length of a record) each time.

Positioning the File Pointer with SEEK

If you want to move the file pointer to a certain place in a file without actually performing any I/O, use the **SEEK** statement. Its syntax is:

SEEK [[#]]*filenumber%*, *position&*

After a **SEEK** statement, the next read or write operation in the file opened with *filenumber%* begins at the byte noted in *position&*.

The counterpart to the **SEEK** statement is the **SEEK** function, with this syntax:

SEEK(*filenumber%*)

The **SEEK** function tells you the byte position where the very next read or write operation begins. (If you are using binary I/O to access a file, the **LOC** and **SEEK** functions give similar results, but **LOC** returns the position of the last byte read or written, while **SEEK** returns the position of the next byte to be read or written.)

The **SEEK** statement and function also work on files opened for sequential or random access. With sequential access, the statement and the function behave in the same way they do with binary access; that is, the **SEEK** statement moves the file pointer to a specific byte position, and the **SEEK** function returns information about the next byte to read or write.

However, if a file is opened for random access, the **SEEK** statement can move the file pointer only to the beginning of a record, not to a byte within a record. Also, when used with random-access files, the **SEEK** function returns the number of the next record rather than the position of the next byte.

Example The following program opens a BASIC Quick library, then reads and prints the names of BASIC procedures and other external symbols contained in the library. This program is in the file named QLBDUMP.BAS on the Microsoft BASIC distribution disks.

```
' This program prints the names of Quick library procedures.
DECLARE SUB DumpSym (SymStart AS INTEGER, QHdrPos AS LONG)

TYPE ExeHdr                         ' Part of DOS .EXE header.
    other1    AS STRING* 8          ' Other header information.
    CParHdr   AS INTEGER            ' Size of header in paragraphs.
    other2    AS STRING* 10         ' Other header information.
    IP        AS INTEGER            ' Initial IP value.
    CS        AS INTEGER            ' Initial (relative) CS value.
END TYPE
TYPE QBHdr                  ' QLB header.
    QBHead    AS STRING* 6          ' QBX specific heading.
    Magic     AS INTEGER            ' Magic word: identifies file as a Quick
                                    ' library.
    SymStart  AS INTEGER            ' Offset from header to first code symbol.
    DatStart  AS INTEGER            ' Offset from header to first data symbol.
END TYPE

TYPE QbSym                  ' QuickLib symbol entry.
    Flags     AS INTEGER            ' Symbol flags.
    NameStart AS INTEGER            ' Offset into name table.
    Other     AS STRING* 4          ' Other header information.
END TYPE

DIM EHdr AS ExeHdr, Qhdr AS QBHdr, QHdrPos AS LONG

INPUT "Enter Quick library filename: ", FileName$
FileName$ = UCASE$(FileName$)
IF INSTR(FileName$,".QLB") = 0 THEN FileName$ = FileName$ + ".QLB"
INPUT "Enter output filename or press Enter for screen: ", OutFile$
OutFile$ = UCASE$(OutFile$)
IF OutFile$ = "" THEN OutFile$ = "CONS:"
OPEN FileName$ FOR BINARY AS #1
OPEN OutFile$ FOR OUTPUT AS #2

GET #1,   , EHdr                    ' Read the EXE format header.
QHdrPos   = (EHdr.CParHdr     + EHdr.CS) * 16 + EHdr.IP + 1

GET #1,   QHdrPos, Qhdr             ' Read the QuickLib format header.
IF Qhdr.Magic <> &H6C75THEN PRINT "Not a QBX UserLibrary": END

PRINT #2, "Code Symbols:": PRINT #2,
DumpSym Qhdr.SymStart, QHdrPos' dump code symbols
PRINT #2,
PRINT #2, "Data Symbols:": PRINT #2, ""
DumpSym Qhdr.DatStart, QHdrPos' dump data symbols
PRINT #2,

END
```

```
SUB DumpSym (SymStart AS INTEGER, QHdrPos AS LONG)
   DIM QlbSym AS QbSym
   DIM NextSym AS LONG, CurrentSym AS LONG

   ' Calculate the location of the first symbol entry,
   ' then read that entry:
   NextSym = QHdrPos + SymStart
   GET #1, NextSym, QlbSym
DO
   NextSym = SEEK(1)    ' Save the location of the next symbol.
   CurrentSym = QHdrPos + QlbSym.NameStart
   SEEK #1, CurrentSym ' Use SEEK to move to the name
                       ' for the current symbol entry.
   Prospect$ = INPUT$(40, 1)' Read the longest legal string,
                       ' plus one additional byte for
                       ' the final null character (CHR$(0)).

   ' Extract the null-terminated name:
   SName$ = LEFT$(Prospect$, INSTR(Prospect$, CHR$(0)))

   ' Print only those names that do not begin with "__", "$", or
   ' "b$" as these names are usually considered reserved:
   T$ = LEFT$(SName$, 2)
   IF T$ <> "__" AND LEFT$(SName$, 1) <> "$" AND UCASE$(T$) <> "B$" THEN
      PRINT #2, "   " + SName$
   END IF

     GET #1, NextSym, QlbSym ' Read a symbol entry.
   LOOP WHILE QlbSym.Flags    ' Flags=0 (false) means end of table.

END SUB
```

Working with Devices

Microsoft BASIC supports device I/O. This means certain computer peripherals can be opened for I/O just like data files on disk. Exceptions to the statements and functions that can be used with these devices are noted in the following section,"Differences Between Device I/O and File I/O." Table 3.1 lists the devices supported by BASIC.

Table 3.1 Devices Supported by BASIC for I/O

Name	Device	I/O mode supported
COM1:	First serial port	Input and output
COM2:	Second serial port	Input and output
CONS:	Screen	Output only

Table 3.1 *Continued*

Name	Device	I/O mode supported
KYBD:	Keyboard	Input only
LPT1:	First printer	Output only
LPT2:	Second printer	Output only
LPT3:	Third printer	Output only
SCRN:	Screen	Output only

Differences Between Device I/O and File I/O

Certain functions and statements used for file I/O are not allowed for device I/O, while other statements and functions have the same name but behave differently. These are some of the differences:

- The CONS, SCRN, and LPT*n* devices cannot be opened in the input, append, or random-access modes.
- The KYBD device cannot be opened in the output, append, or random-access modes.
- The **EOF**, **LOC**, and **LOF** functions cannot be used with the CONS, KYBD, LPT*n*, or SCRN devices.
- The **EOF**, **LOC**, and **LOF** functions can be used with the COM*n* serial device; however, the values these functions return have a different meaning than the values they return when used with data files. (See the next section for an explanation of what these functions do when used with COM*n*.)

Example The following program shows how the devices LPT1 or SCRN can be opened for output using the same syntax as that for data files. This program reads all the lines from the file chosen by the user and then prints the lines on the screen or the printer according to the user's choice.

```
CLS
' Input the name of the file to look at:
INPUT "Name of file to display: ", FileNam$

' If no name is entered, end the program;
' otherwise, open the given file for reading (INPUT):
IF FileNam$ = "" THEN END ELSE OPEN FileNam$ FOR INPUT AS #1

' Input choice for displaying file (Screen or Printer);
' continue prompting for input until either the "S" or "P"
' key is pressed:
DO
   ' Go to row 2, column 1 on the screen and print prompt:
   LOCATE 2, 1, 1
   PRINT "Send output to screen (S), or to printer (P): ";
```

```
    ' Print over anything after the prompt:
    PRINT SPACE$(2);

    ' Relocate cursor after the prompt, and make it visible:
    LOCATE 2, 47, 1
    Choice$ = UCASE$(INPUT$(1))      ' Get input.
    PRINT Choice$
LOOP WHILE Choice$ <> "S" AND Choice$ <> "P"

' Depending on the key pressed, open either the printer
' or the screen for output:
SELECT CASE Choice$
   CASE "P"
      OPEN "LPT1:" FOR OUTPUT AS #2
      PRINT "Printing file on printer."
   CASE "S"
      CLS
      OPEN "SCRN:" FOR OUTPUT AS #2
END SELECT

' Set the width of the chosen output device to 80 columns:
WIDTH #2, 80

' As long as there are lines in the file, read a line
' from the file and print it on the chosen device:
DO UNTIL EOF(1)
   LINE INPUT #1, LineBuffer$
   PRINT #2, LineBuffer$
LOOP

CLOSE      ' End input from the file and output to the device.
END
```

Communications Through the Serial Port

The **OPEN "COM*n*:"** statement (where *n* can be 1 or, if you have two serial ports, 2) allows you to open your computer's serial port(s) for serial (bit-by-bit) communication with other computers or with peripheral devices such as modems or serial printers. The following are some of the parameters you can specify:

- Rate of data transmission, measured in "baud" (bits per second)
- Whether or not to detect transmission errors and how those errors will be detected
- How many stop bits (1, 1.5, or 2) are to be used to signal the end of a transmitted byte
- How many bits in each byte of data transmitted or received constitute actual data

When the serial port is opened for communication, an input buffer is set aside to hold the bytes being read from another device. This is because, at high baud rates, characters arrive faster than they can be processed. The default size for this buffer is 512 bytes, and it can be modified with the **LEN** = *reclen%* option of the **OPEN "COM***n***:"** statement. The values returned by the **EOF**, **LOC**, and **LOF** functions when used with a communications device return information about the size of this buffer, as shown in the following table:

Function	Information returned
EOF	Whether any characters are waiting to be read from the input buffer
LOC	The number of characters waiting in the input buffer
LOF	The amount of space remaining (in bytes) in the output buffer

Since every character is potentially significant data, **INPUT #** and **LINE INPUT #** have serious drawbacks for getting input from another device. This is because **INPUT #** stops reading data into a variable when it encounters a comma or new-line character (and, sometimes, a space or double quotation mark), and **LINE INPUT #** stops reading data when it encounters a new-line character. This makes **INPUT$** the best function to use for input from a communications device, since it reads all characters.

The following line uses the **LOC** function to check the input buffer for the number of characters waiting there from the communications device opened as file #1; it then uses the **INPUT$** function to read those characters, assigning them to a string variable named `ModemInput$`:

```
ModemInput$ = INPUT$(LOC(1), #1)
```

Sample Applications

The sample applications listed in this section include a screen-handling program that prints a calendar for any month in any year from 1899 to 2099, a file I/O program that builds and searches an index of record numbers from a random-access file, and a communications program that makes your PC behave like a terminal when connected to a modem.

Perpetual Calendar (CAL.BAS)

After prompting the user to input a month from 1 to 12 and a year from 1899 to 2099, the following program prints the calendar for the given month and year. The `IsLeapYear` procedure makes appropriate adjustments to the calendar for months in a leap year.

Statements and Functions Used

This program demonstrates the following screen-handling statements and functions:

- **INPUT**
- **INPUT$**
- **LOCATE**
- **POS(0)**
- **PRINT**
- **PRINT USING**
- **TAB**

Program Listing

```
DEFINT A-Z      ' Default variable type is integer.

' Define a data type for the names of the months and the
' number of days in each:
TYPE MonthType
   Number AS INTEGER      ' Number of days in the month.
   MName AS STRING * 9    ' Name  of the month.
END TYPE

' Declare procedures used:
DECLARE FUNCTION IsLeapYear% (N%)
DECLARE FUNCTION GetInput% (Prompt$, Row%, LowVal%, HighVal%)

DECLARE SUB PrintCalendar (Year%, Month%)
DECLARE SUB ComputeMonth (Year%, Month%, StartDay%, TotalDays%)

DIM MonthData(1 TO 12)    AS MonthType

' Initialize month definitions from DATA statements below:
FOR I = 1 TO 12
   READ MonthData(I).MName, MonthData(I).Number
NEXT

' Main loop, repeat for as many months as desired:
DO
   CLS
```

```
   ' Get year and month as input:
   Year = GetInput("Year (1899 to 2099): ", 1, 1899, 2099)
   Month = GetInput("Month (1 to 12): ", 2, 1, 12)

   ' Print the calendar:
   PrintCalendar Year, Month
' Another Date?
   LOCATE 13, 1          ' Locate in 13th row, 1st column.
   PRINT "New Date? ";   ' Keep cursor on same line.
   LOCATE , , 1, 0, 13   ' Turn cursor on and make it one
                         ' character high.
   Resp$ = INPUT$(1)     ' Wait for a key press.
   PRINT Resp$           ' Print the key pressed.
LOOP WHILE UCASE$(Resp$) = "Y"
END

' Data for the months of a year:
DATA January, 31, February, 28,  March, 31
DATA April, 30,   May, 31, June, 30, July, 31, August, 31
DATA September,   30, October, 31, November, 30, December, 31

' ====================== ComputeMonth ======================
'  Computes the first day and the total days in a month
' ==========================================================
'
SUB ComputeMonth (Year, Month, StartDay, TotalDays) STATIC
   SHARED MonthData() AS MonthType

   CONST LEAP = 366 MOD 7
   CONST NORMAL = 365 MOD 7

   ' Calculate total number of days (NumDays) since 1/1/1899:

' Start with whole years:
NumDays = 0
FOR I = 1899 TO Year - 1
    IF IsLeapYear(I) THEN           ' If leap year,
       NumDays = NumDays + LEAP     ' add 366 MOD 7.
    ELSE                            ' If normal year,
       NumDays = NumDays + NORMAL   ' add 365 MOD 7.
    END IF
NEXT
```

```
' Next, add in days from whole months:
FOR I = 1 TO Month - 1
    NumDays = NumDays + MonthData(I).Number
NEXT

' Set the number of days in the requested month:
TotalDays = MonthData(Month).Number

' Compensate if requested year is a leap year:
IF IsLeapYear(Year) THEN

    ' If after February, add one to total days:
    IF Month > 2 THEN
       NumDays = NumDays + 1

    ' If February, add one to the month's days:
    ELSEIF Month = 2 THEN
       TotalDays = TotalDays + 1
    END IF
  END IF

  ' 1/1/1899 was a Sunday, so calculating "NumDays MOD 7"
  ' gives the day of week (Sunday = 0, Monday = 1, Tuesday
  ' = 2, and so on) for the first day of the input month:
  StartDay = NumDays MOD 7
END SUB

' ======================== GetInput ========================
'  Prompts for input, then tests for a valid range
' ==========================================================
'
FUNCTION GetInput (Prompt$, Row, LowVal, HighVal) STATIC

' Locate prompt at specified row, turn cursor on and
' make it one character high:
LOCATE Row, 1, 1, 0, 13
PRINT Prompt$;

' Save column position:
Column = POS(0)

' Input value until it's within range:
```

```
    DO
        LOCATE Row, Column     ' Locate cursor at end of prompt.
        PRINT SPACE$(10)       ' Erase anything already there.
        LOCATE Row, Column     ' Relocate cursor at end of prompt.
        INPUT "", Value        ' Input value with no prompt.
    LOOP WHILE (Value < LowVal OR Value > HighVal)

    ' Return valid input as value of function:
    GetInput = Value

END FUNCTION

' ====================== IsLeapYear =======================
'   Determines if a year is a leap year or not
' ========================================================
'
FUNCTION IsLeapYear (N) STATIC

    ' If the year is evenly divisible by 4 and not divisible
    ' by 100, or if the year is evenly divisible by 400,
    ' then it's a leap year:
    IsLeapYear = (N MOD 4 = 0 AND N MOD 100 <> 0) OR (N MOD 400 = 0)
END FUNCTION

' ====================== PrintCalendar ====================
'   Prints a formatted calendar given the year and month
' ========================================================
'
SUB PrintCalendar (Year, Month) STATIC
SHARED MonthData() AS MonthType

    ' Compute starting day (Su M Tu ...)
    ' and total days for the month:
    ComputeMonth Year, Month, StartDay, TotalDays
    CLS
    Header$ = RTRIM$(MonthData(Month).MName) + "," + STR$(Year)

    ' Calculate location for centering month and year:
    LeftMargin = (35 - LEN(Header$)) \ 2
' Print header:
    PRINT TAB(LeftMargin); Header$
    PRINT
    PRINT "Su    M    Tu    W    Th    F    Sa"
    PRINT
```

```
' Recalculate and print tab
' to the first day of the month (Su M Tu ...):
LeftMargin = 5 * StartDay + 1
PRINT TAB(LeftMargin);

' Print out the days of the month:
FOR I = 1 TO TotalDays
    PRINT USING "##_   "; I;

    ' Advance to the next line
    ' when the cursor is past column 32:
    IF POS(0) > 32 THEN PRINT
NEXT

END SUB
```

Output

```
            August, 1989

Su     M    Tu     W    Th     F    Sa

                1     2     3     4     5
 6     7     8     9    10    11    12
13    14    15    16    17    18    19
20    21    22    23    24    25    26
27    28    29    30    31


New Date?
```

Indexing a Random-Access File (INDEX.BAS)

The following program uses an indexing technique to store and retrieve records in a random-access file. Each element of the `Index()` array has two parts: a string field (`PartNumber`) and an integer field (`RecordNumber`). This array is sorted alphabetically on the `PartNumber` field, which allows the array to be rapidly searched for a specific part number using a binary search.

The `Index` array functions much like the index to a book. When you want to find the pages in a book that deal with a particular topic, you look up an entry for that topic in the index. The entry then points to a page number in the book. Similarly, this program looks up a part number in the alphabetically sorted `Index()` array. Once it finds the part number, the associated record number in the `RecordNumber` field points to the record containing all the information for that part.

Statements and Functions Used

This program demonstrates the following statements and functions used in accessing random-access files:

- **TYPE...END TYPE**
- **OPEN...FOR RANDOM**
- **GET #**
- **PUT #**
- **LOF**

Program Listing

```
DEFINT A-Z

' Define the symbolic constants used globally in the program:
CONST FALSE = 0, TRUE = NOT FALSE

' Define a record structure for random-access records:
TYPE StockItem
   PartNumber AS STRING * 6
   Description AS STRING * 20
   UnitPrice AS SINGLE
   Quantity AS INTEGER
END TYPE

' Define a record structure for each element of the index:
TYPE IndexType
   RecordNumber AS INTEGER
   PartNumber AS STRING * 6
END TYPE

' Declare procedures that will be called:
DECLARE FUNCTION Filter$ (Prompt$)
DECLARE FUNCTION FindRecord% (PartNumber$, RecordVar AS StockItem)
```

```
DECLARE SUB AddRecord (RecordVar AS StockItem)
DECLARE SUB InputRecord (RecordVar AS StockItem)
DECLARE SUB PrintRecord (RecordVar AS StockItem)
DECLARE SUB SortIndex ()
DECLARE SUB ShowPartNumbers ()
' Define a buffer (using the StockItem type)
' and define and dimension the index array:
DIM StockRecord AS StockItem, Index(1 TO 100) AS IndexType

' Open the random-access file:
OPEN "STOCK.DAT" FOR RANDOM AS #1 LEN = LEN(StockRecord)

' Calculate number of records in the file:
NumberOfRecords = LOF(1) \ LEN(StockRecord)

' If there are records, read them and build the index:
IF NumberOfRecords <> 0 THEN
   FOR RecordNumber = 1 TO NumberOfRecords

      ' Read the data from a new record in the file:
      GET #1, RecordNumber, StockRecord

      ' Place part number and record number in index:
      Index(RecordNumber).RecordNumber = RecordNumber
      Index(RecordNumber).PartNumber = StockRecord.PartNumber
   NEXT

   SortIndex              ' Sort index in part-number order.
END IF

DO                        ' Main-menu loop.
   CLS
   PRINT "(A)dd records."
   PRINT "(L)ook up records."
   PRINT "(Q)uit program."
   PRINT
   LOCATE , , 1
   PRINT "Type your choice (A, L, or Q) here: ";

   ' Loop until user presses, A, L, or Q:
   DO
      Choice$ = UCASE$(INPUT$(1))
   LOOP WHILE INSTR("ALQ", Choice$) = 0
```

```
' Branch according to choice:
SELECT CASE Choice$
   CASE "A"
 AddRecord StockRecord
   CASE "L"
 IF NumberOfRecords = 0 THEN
    PRINT : PRINT "No records in file yet. ";
    PRINT "Press any key to continue.";
    Pause$ = INPUT$(1)
 ELSE
    InputRecord StockRecord
 END IF
   CASE "Q"            ' End program.
END SELECT
LOOP UNTIL Choice$ = "Q"

CLOSE #1                    ' All done, close file and end.
END

' ======================== AddRecords ======================
' Adds records to the file from input typed at the keyboard
' ==========================================================
'
SUB AddRecord (RecordVar AS StockItem) STATIC
   SHARED Index() AS IndexType, NumberOfRecords
   DO
      CLS
      INPUT "Part Number: ", RecordVar.PartNumber
      INPUT "Description: ", RecordVar.Description

      ' Call the Filter$ function to input price & quantity:
      RecordVar.UnitPrice = VAL(Filter$("Unit Price : "))
      RecordVar.Quantity = VAL(Filter$("Quantity   : "))

      NumberOfRecords = NumberOfRecords + 1

      PUT #1, NumberOfRecords, RecordVar
```

```
        Index(NumberOfRecords).RecordNumber = NumberOfRecords
        Index(NumberOfRecords).PartNumber = RecordVar.PartNumber
        PRINT : PRINT "Add another? ";
        OK$ = UCASE$(INPUT$(1))
    LOOP WHILE OK$ = "Y"

    SortIndex                ' Sort index file again.
END SUB

' ========================= Filter$ =======================
' Filters all non-numeric characters from a string
' and returns the filtered string
' =========================================================
'
FUNCTION Filter$ (Prompt$) STATIC
    ValTemp2$ = ""
    PRINT Prompt$;                          ' Print the prompt passed.
    INPUT "", ValTemp1$                     ' Input a number as
                                            ' a string.
    StringLength = LEN(ValTemp1$)           ' Get the string's length.
    FOR I% = 1 TO StringLength              ' Go through the string,
        Char$ = MID$(ValTemp1$, I%, 1) ' one character at a time.

        ' Is the character a valid part of a number (i.e.,
        ' a digit or a decimal point)?  If yes, add it to
        ' the end of a new string:
        IF INSTR(".0123456789", Char$) > 0 THEN
            ValTemp2$ = ValTemp2$ + Char$

        ' Otherwise, check to see if it's a lowercase "l",
        ' since typewriter users may enter a one that way:
        ELSEIF Char$ = "l" THEN
            ValTemp2$ = ValTemp2$ + "1" ' Change the "l" to a "1."
        END IF
    NEXT I%

    Filter$ = ValTemp2$                     ' Return filtered string.

END FUNCTION
```

```
' ======================= FindRecord% ==================
'  Uses a binary search to locate a record in the index
' ======================================================
'
FUNCTION FindRecord% (Part$, RecordVar AS StockItem) STATIC
   SHARED Index() AS IndexType, NumberOfRecords

   ' Set top and bottom bounds of search:
   TopRecord = NumberOfRecords
   BottomRecord = 1

   ' Search until top of range is less than bottom:
   DO UNTIL (TopRecord < BottomRecord)

   ' Choose midpoint:
   Midpoint = (TopRecord + BottomRecord) \ 2

   ' Test to see if it's the one wanted (RTRIM$()
   ' trims trailing blanks from a fixed string):
   Test$ = RTRIM$(Index(Midpoint).PartNumber)

   ' If it is, exit loop:
   IF Test$ = Part$ THEN
EXIT DO

   ' Otherwise, if what you're looking for is greater,
   ' move bottom up:
   ELSEIF Part$ > Test$ THEN
BottomRecord = Midpoint + 1

   ' Otherwise, move the top down:
   ELSE
TopRecord = Midpoint - 1
   END IF
LOOP

' If part was found, input record from file using
' pointer in index and set FindRecord% to TRUE:
IF Test$ = Part$ THEN
   GET #1, Index(Midpoint).RecordNumber, RecordVar
   FindRecord% = TRUE
```

```
    ' Otherwise, if part was not found, set FindRecord%
    ' to FALSE:
    ELSE
        FindRecord% = FALSE
    END IF
END FUNCTION

' ====================== InputRecord ====================
'    First, InputRecord calls ShowPartNumbers, which prints
'    a menu of part numbers on the top of the screen. Next,
'    InputRecord prompts the user to enter a part number.
'    Finally, it calls the FindRecord and PrintRecord
'    procedures to find and print the given record.
' =======================================================
'
SUB InputRecord (RecordVar AS StockItem) STATIC
    CLS
    ShowPartNumbers       ' Call the ShowPartNumbers SUB procedure.

    ' Print data from specified records
    ' on the bottom part of the screen:
    DO
        PRINT "Type a part number listed above ";
        INPUT "(or Q to quit) and press <ENTER>: ", Part$
        IF UCASE$(Part$) <> "Q" THEN
     IF FindRecord(Part$, RecordVar) THEN
         PrintRecord RecordVar
     ELSE
         PRINT "Part not found."
     END IF
        END IF
        PRINT STRING$(40, "_")
    LOOP WHILE UCASE$(Part$) <> "Q"

    VIEW PRINT    ' Restore the text viewport to entire screen.
END SUB
```

```
' ======================= PrintRecord ====================
'                Prints a record on the screen
' ========================================================
'
SUB PrintRecord (RecordVar AS StockItem) STATIC
   PRINT "Part Number: "; RecordVar.PartNumber
   PRINT "Description: "; RecordVar.Description
   PRINT USING "Unit Price :$$###.##"; RecordVar.UnitPrice
   PRINT "Quantity   :"; RecordVar.Quantity
END SUB

' ===================== ShowPartNumbers ==================
' Prints an index of all the part numbers in the upper part
' of the screen
' ========================================================
'
SUB ShowPartNumbers STATIC
   SHARED index() AS IndexType, NumberOfRecords

   CONST NUMCOLS = 8, COLWIDTH = 80 \ NUMCOLS

   ' At the top of the screen, print a menu indexing all
   ' the part numbers for records in the file. This menu is
   ' printed in columns of equal length (except possibly the
   ' last column, which may be shorter than the others):
   ColumnLength = NumberOfRecords
   DO WHILE ColumnLength MOD NUMCOLS
      ColumnLength = ColumnLength + 1
   LOOP
   ColumnLength = ColumnLength \ NUMCOLS
   Column = 1
   RecordNumber = 1
DO UNTIL RecordNumber > NumberOfRecords
   FOR Row = 1 TO ColumnLength
 LOCATE Row, Column
 PRINT index(RecordNumber).PartNumber
 RecordNumber = RecordNumber + 1
 IF RecordNumber > NumberOfRecords THEN EXIT FOR
   NEXT Row
   Column = Column + COLWIDTH
LOOP
```

```
    LOCATE ColumnLength + 1, 1
    PRINT STRING$(80, "_")        ' Print separator line.

    ' Scroll information about records below the part-number
    ' menu (this way, the part numbers are not erased):
    VIEW PRINT ColumnLength + 2 TO 24
END SUB

' ========================== SortIndex ====================
'                 Sorts the index by part number
' =========================================================
'
SUB SortIndex STATIC
SHARED Index() AS IndexType, NumberOfRecords

' Set comparison offset to half the number of records
' in Index:
Offset = NumberOfRecords \ 2

' Loop until offset gets to zero:
DO WHILE Offset > 0
    Limit = NumberOfRecords - Offset
    DO

 ' Assume no switches at this offset:
 Switch = FALSE

 ' Compare elements and switch ones out of order:
 FOR I = 1 TO Limit
     IF Index(I).PartNumber > Index(I + Offset).PartNumber THEN
         SWAP Index(I), Index(I + Offset)
         Switch = I
     END IF
     NEXT I

     ' Sort on next pass only to where
     ' last switch was made:
     Limit = Switch
        LOOP WHILE Switch

        ' No switches at last offset, try one half as big:
        Offset = Offset \ 2
    LOOP
END SUB
```

Output

```
0235      0417      0583      8721

Type a part number listed above (or Q to quit) and press <ENTER>: 0417
Part Number: 0417
Description: oil filter
Unit Price :   $5.99
Quantity   : 12

Type a part number listed above (or Q to quit) and press <ENTER>: 8721
Part Number: 8721
Description: chamois cloth
Unit Price :   $8.99
Quantity   : 23

Type a part number listed above (or Q to quit) and press <ENTER>:
```

Terminal Emulator (TERMINAL.BAS)

The following simple program turns your computer into a "dumb" terminal; that is, it makes your computer function solely as an I/O device in tandem with a modem. This program uses the **OPEN "COM1:"** statement and associated device I/O functions to do the following things:

- Send the characters you type to the modem
- Print characters returned by the modem on the screen

Note that typed characters displayed on the screen are first sent to the modem and then returned to your computer through the open communications channel. You can verify this if you have a modem with Receive Detect (RD) and Send Detect (SD) lights—they will flash in sequence as you type.

To dial a number, you would have to enter the Attention Dial Touch-Tone (ATDT) or Attention Dial Pulse (ATDP) commands at the keyboard (assuming you have a Hayes-compatible modem).

Any other commands sent to the modem would also have to be entered at the keyboard.

Statements and Functions Used

This program demonstrates the following statements and functions used in communicating with a modem through your computer's serial port:

- **OPEN "COM1:"**
- **EOF**
- **INPUT$**
- **LOC**

Program Listing

```
DEFINT A-Z

DECLARE   SUB Filter (InString$)

COLOR 7, 1                ' Set screen color.
CLS

Quit$ = CHR$(0) + CHR$(16)    ' Value returned by INKEY$
                              ' when Alt+Q is pressed.

' Set up prompt on bottom line of screen and turn cursor on:
LOCATE 24, 1, 1
PRINT STRING$(80, "_");
LOCATE 25, 1
PRINT TAB(30); "Press Alt+Q to quit";

VIEW PRINT 1 TO 23            ' Print between lines 1 & 23.

' Open communications (1200 baud, no parity, 8-bit data,
' 1 stop bit, 256-byte input buffer):
OPEN "COM1:1200,N,8,1" FOR RANDOM AS #1    LEN = 256

DO                                         ' Main communications loop.

   KeyInput$ = INKEY$                      ' Check the keyboard.

   IF KeyInput$    = Quit$    THEN   ' Exit the loop if the user
      EXIT DO                       ' pressed Alt+Q.
```

```
    ELSEIF KeyInput$ <> "" THEN          ' Otherwise, if the user has
       PRINT #1, KeyInput$;              ' pressed a key, send the
    END IF                               ' character typed to modem.
  ' Check the modem. If characters are waiting (EOF(1) is
  ' true), get them and print them to the screen:
  IF NOT EOF(1) THEN

       ' LOC(1) gives the number of characters waiting:
       ModemInput$ = INPUT$(LOC(1), #1)

       Filter ModemInput$                ' Filter out line feeds and
       PRINT ModemInput$;                ' backspaces, then print.
    END IF
LOOP

CLOSE                                    ' End communications.
CLS
END
'
' ======================== Filter =======================
'               Filters characters in an input string
' =======================================================
'
SUB Filter (InString$) STATIC

    ' Look for Backspace characters and recode
    ' them to CHR$(29) (the Left direction key):
    DO
       BackSpace = INSTR(InString$, CHR$(8))
       IF BackSpace THEN
       MID$(InString$, BackSpace) = CHR$(29)
       END IF
    LOOP WHILE BackSpace

    ' Look for line-feed characters and
    ' remove any found:
    DO
       LnFd = INSTR(InString$, CHR$(10))
       IF LnFd THEN
    InString$=LEFT$(InString$,LnFd-1)+MID$(InString$,LnFd+1)
       END IF
    LOOP WHILE LnFd

END SUB
```

Chapter 4
String Processing

This chapter shows you how to manipulate sequences of ASCII characters, known as strings. String manipulation is indispensable when you are processing text files, sorting data, or modifying string-data input.

When you are finished with this chapter, you will know how to perform the following string-processing tasks:

- Declare fixed-length strings.
- Compare strings and sort them alphabetically.
- Search for one string inside another.
- Get part of a string.
- Trim spaces from the beginning or end of a string.
- Generate a string by repeating a single character.
- Change uppercase letters in a string to lowercase and vice versa.
- Convert numeric expressions to string representations and vice versa.

Strings Defined

A string is a sequence of contiguous characters. Examples of characters are the letters of the alphabet (a–z and A–Z), punctuation symbols such as commas (,) or question marks (?), and other symbols from the fields of math and finance such as plus (+) or percent (%) signs.

In this chapter, the term "string" can refer to any of the following:

- A string constant
- A string variable
- A string expression

String constants are declared in one of two ways:

- By enclosing a sequence of characters between double quotation marks, as in the following **PRINT** statement:

```
PRINT "Processing the file. Please wait."
```

  This is known as a "literal string constant."

- By setting a name equal to a literal string in a **CONST** statement, as in the following:

```
' Define the string constant MESSAGE:
CONST MESSAGE = "Drive door open."
```

  This is known as a "symbolic string constant."

String variables can be declared in one of three ways:

- By appending the string-type suffix ($) to the variable name:

```
LINE INPUT "Enter your name: "; Buffer$
```

- By using the **DEFSTR** statement:

```
' All variables beginning with the letter "B"
' are strings, unless they end with one of the
' numeric-type suffixes (%, &, !, @, or #):
DEFSTR B
.
.
.
Buffer = "Your name here"   ' Buffer is a string variable.
```

- By declaring the string name in an **AS STRING** statement:

```
DIM Buffer1 AS STRING       ' Buffer1 is a variable-
                            ' length string.
DIM Buffer2 AS STRING*10    ' Buffer2 is a fixed-length
                            ' string 10 bytes long.
```

A string expression is a combination of string variables, constants, and/or string functions.

Of course, the character components of strings are not stored in your computer's memory in a form generally recognizable. Instead, each character is translated to a number known as the American Standard Code for Information Interchange code for that character. For example, uppercase "A" is stored as decimal 65 (or hexadecimal 41H), while lowercase "a" is stored as decimal 97 (or hexadecimal 61H).

You can also use the BASIC **ASC** function to determine the ASCII code for a character; for example, `ASC("A")` returns the value 65. The inverse of the **ASC** function is the **CHR$** function. **CHR$** takes an ASCII code as its argument and returns the character with that code; for example, the statement `PRINT CHR$(64)` displays the character @.

Variable- and Fixed-Length Strings

In previous versions of BASIC, strings were always variable length. BASIC now supports variable-length strings and fixed-length strings.

Variable-Length Strings

Variable-length strings are "elastic"; that is, they expand and contract to store different numbers of characters (from none to a maximum of 32,767).

Example

The length of the variable `Temp$` in the following example varies according to the size of what is stored in `Temp$`:

```
Temp$ = "1234567"
' LEN function returns length of string
' (number of characters it contains):
PRINT LEN(Temp$)

Temp$ = "1234"
PRINT LEN(Temp$)
```

Output

```
7
4
```

Fixed-Length Strings

Fixed-length strings are commonly used as record elements in a **TYPE...END TYPE** user-defined data type. However, they can also be declared by themselves in **COMMON, DIM, REDIM, SHARED,** or **STATIC** statements, as in the following statement:

```
DIM Buffer AS STRING * 10
```

Examples

As their name implies, fixed-length strings have a constant length, regardless of the length of the string stored in them. This is shown by the output from the following example:

```
DIM LastName AS STRING * 12
DIM FirstName AS STRING * 10

LastName = "Huntington-Ashley"
FirstName = "Claire"

PRINT "12345678901234567890123456789O"
PRINT FirstName; LastName
PRINT LEN(FirstName)
PRINT LEN(LastName)
```

Output

```
12345678901234567890123456789O
Claire    Huntington-A
 10
 12
```

Note that the lengths of the string variables `FirstName` and `LastName` did not change, as demonstrated by the values returned by the **LEN** function (as well as the four spaces following the six-letter name, `Claire`).

The output from the preceding example also illustrates how values assigned to fixed-length variables are left-justified and, if necessary, truncated on the right. It is not necessary to use the **LSET** function to left-justify values in fixed-length strings; this is done automatically. If you want to right-justify a string inside a fixed-length string, use **RSET**, as shown here:

```
DIM NameBuffer AS STRING * 10
RSET NameBuffer = "Claire"
PRINT "1234567890"
PRINT NameBuffer
```

Output

```
1234567890
    Claire
```

Combining Strings

Two strings can be combined with the addition operator (+). The string following the plus operator is appended to the string preceding the plus operator.

Examples

The following example combines two strings:

```
A$ = "first string"
B$ = "second string"
C$ = A$ + B$
PRINT C$
```

Output

```
first stringsecond string
```

The process of combining strings in this way is called "concatenation," which means linking together.

Note that in the previous example, the two strings are combined without any intervening spaces. If you want a space in the combined string, you could pad one of the strings `A$` or `B$`, like this:

```
B$ = " second string"          ' Leading blank in B$
```

Because values are left-justified in fixed-length strings, you may find extra spaces when you concatenate them, as in this example:

```
TYPE NameType
   First AS STRING * 10
   Last  AS STRING * 12
END TYPE

DIM NameRecord AS NameType

' The constant "Ed" is left-justified in the variable
' NameRecord.First, so there are eight trailing blanks:
NameRecord.First = "Ed"
NameRecord.Last  = "Feldstein"

' Print a line of numbers for reference:
PRINT "12345678901234567890123456789O"

WholeName$ = NameRecord.First + NameRecord.Last
PRINT WholeName$
```

Output

```
12345678901234567890123456789O
Ed        Feldstein
```

The **LTRIM$** function returns a string with its leading spaces stripped away, while the **RTRIM$** function returns a string with its trailing spaces stripped away. The original string is unaltered. (See the section "Retrieving Parts of Strings" later in this chapter for more information on these functions.)

Comparing Strings

Strings are compared with the following relational operators:

Operator	Meaning
<>	Not equal
=	Equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

A single character is greater than another character if its ASCII value is greater. For example, the ASCII value of the letter "B" is greater than the ASCII value of the letter "A," so the expression `"B" > "A"` is true.

When comparing two strings, BASIC looks at the ASCII values of corresponding characters. The first character where the two strings differ determines the alphabetical order of the strings. For instance, the strings "doorman" and "doormats" are the same up to the seventh character in each ("n" and "t"). Since the ASCII value of "n" is less than the ASCII value of "t," the expression `"doorman" < "doormats"` is true. Note that the ASCII values for letters follow alphabetical order from A to Z and from a to z, so you can use the relational operators listed in the preceding table to alphabetize strings. Moreover, uppercase letters have smaller ASCII values than lowercase letters, so in a sorted list of strings, "ASCII" would come before "ascii."

If there is no difference between corresponding characters of two strings and they are the same length, then the two strings are equal. If there is no difference between corresponding characters of two strings, but one of the strings is longer, then the longer string is greater than the shorter string. For example `abc =abc` and `aaaaaaa >aaa` are true expressions.

Leading and trailing blank spaces are significant in string comparisons. For example, the string " test" is less than the string "test," since a blank space (" ") is less than a "t"; on the other hand, the string "test " is greater than the string "test." Keep this in mind when comparing fixed-length and variable-length strings, since fixed-length strings may contain trailing spaces.

Searching for Strings

One of the most common string-processing tasks is searching for a string inside another string. The **INSTR**(*stringexpression1$*, *stringexpression2$*) function tells you whether or not *string2* is contained in *stringexpression1$* by returning the position of the first character in *stringexpression1$* (if any) where the match begins.

Examples

The following example finds the starting position of one string inside another string:

```
String1$ = "A line of text with 37 letters in it."
String2$ = "letters"

PRINT "         1         2         3         4"
PRINT "12345678901234567890123456789012345678901"
PRINT String1$
PRINT String2$
PRINT INSTR(String1$, String2$)
```

Output

```
1         2         3         4
12345678901234567890123456789012345678901
A line of text with 37 letters in it.
letters
 24
```

If no match is found (that is, *stringexpression2$* is not a substring of *stringexpression1$*), **INSTR** returns the value 0.

A variation of the preceding syntax, **INSTR**(*start%*, *stringexpression1$*, *stringexpression2$*), allows you to specify where you want the search to start in *stringexpression1$*. To illustrate, making the following modification to the preceding example changes the output:

```
' Start looking for a match at the 30th
' character of String1$:
PRINT INSTR(30, String1$, String2$)
```

Output

```
1         2         3         4
12345678901234567890123456789012345678901234567890
A line of text with 37 letters in it.
letters
 0
```

In other words, the string `letters` does not appear after the 30th character of `String1$`.

The **INSTR**(*position*, *string1*, *string2*) variation is useful for finding every occurrence of *stringexpression2$* in *stringexpression1$* instead of just the first occurrence, as shown in the next example:

```
String1$ = "the dog jumped over the broken saxophone."
String2$ = "the"
PRINT String1$

Start      = 1
NumMatches = 0

DO
   Match = INSTR(Start, String1$, String2$)
   IF Match > 0 THEN
      PRINT TAB(Match); String2$
      Start      = Match + 1
      NumMatches = NumMatches + 1
   END IF
LOOP WHILE MATCH

PRINT "Number of matches ="; NumMatches
```

Output

```
the dog jumped over the broken saxophone.
the
                      the
Number of matches = 2
```

Retrieving Parts of Strings

The section "Combining Strings" shows how to put strings together (concatenate them) by using the addition operator (+). Several functions are available in BASIC that take strings apart, returning pieces of a string, either from the left side, the right side, or the middle of a target string.

Retrieving Characters from the Left Side of a String

The **LEFT$** and **RTRIM$** functions return characters from the left side of a string. The **LEFT$**(*stringexpression$*, *n%*) function returns *number* characters from the left side of *string*.

Examples

The following example retrieves four characters from the left side of a string:

```
Test$ = "Overtly"

' Print the leftmost 4 characters from Test$:
PRINT LEFT$(Test$, 4)
```

Output

```
Over
```

Note that **LEFT$**, like the other functions described in this chapter, does not change the original string `Test$`; it merely returns a different string, a copy of part of `Test$`.

The **RTRIM$** function returns the left part of a string after removing any blank spaces at the end. For example, contrast the output from the two **PRINT** statements in the following example:

```
PRINT "a left-justified string      "; "next"
PRINT RTRIM$("a left-justified string      "); "next"
```

Output

```
a left-justified string      next
a left-justified stringnext
```

The **RTRIM$** function is useful for comparing fixed-length and variable-length strings, as in the next example:

```
DIM NameRecord AS STRING * 10
NameRecord = "Ed"

NameTest$ = "Ed"
```

```
' Use RTRIM$ to trim all the blank spaces from the right
' side of the fixed-length string NameRecord, then compare
' it with the variable-length string NameTest$:
IF RTRIM$(NameRecord) = NameTest$ THEN
   PRINT "They're equal"
ELSE
   PRINT "They're not equal"
END IF
```

Output

```
They're equal
```

Retrieving Characters from the Right Side of a String

The **RIGHT$** and **LTRIM$** functions return characters from the right side of a string. The **RIGHT$**(*stringexpression$*, *n%*) function returns *n%* characters from the right side of *stringexpression$*.

Examples

The following example retrieves five characters from the right side of a string:

```
Test$ = "1234567890"

' Print the rightmost 5 characters from Test$:
PRINT RIGHT$(Test$,5)
```

Output

```
67890
```

The **LTRIM$** function returns the right part of a string after removing any blank spaces at the beginning. For example, contrast the output from the next two **PRINT** statements:

```
PRINT "first"; "      a right-justified string"
PRINT "first"; LTRIM$("      a right-justified string")
```

Output

```
first      a right-justified string
firsta right-justified string
```

Retrieving Characters from Anywhere in a String

Use the **MID$** function to retrieve any number of characters from any point in a string. The **MID$**(*stringexpression$*, *start%*, *length%*) function returns *n%* characters from *stringexpression$*, starting at the character with position *start%*. For example, the following statement starts at the 12th character of the string ("h") and returns the next three characters ("hill"):

```
MID$("**over the hill**", 12, 4)
```

Example The following example shows how to use the **MID$** function to step through a line of text character by character:

```
.
.
.
' Get the number of characters in the string of text:
Length = LEN(TextString$)

FOR I = 1 TO Length

  ' Get the first character, then the second, third,
  ' and so forth, up to the end of the string:
  Char$ = MID$(TextString$, I, 1)

  ' Evaluate that character:
  .
  .
  .
NEXT
```

Generating Strings

The easiest way to create a string of one character repeated over and over is to use the intrinsic function **STRING$**. The **STRING$**(*m%*, *stringexpression$*) function produces a new string with *m%* characters, each character of which is the first character of *stringexpression$*. For example, the following statement generates a string of 27 asterisks:

```
Filler$ = STRING$(27, "*")
```

Example For characters that cannot be produced by typing, such as characters whose ASCII values are greater than 127, use the alternative form of this function, **STRING$**(*m%*, *n%*). This form creates a string with *m%* characters, each character of which has the ASCII code specified by *m%*, as in the next example:

```
' Print a string of 10 "@" characters
' (64 is ASCII code for @):
PRINT STRING$(10, 64)
```

Output

```
@@@@@@@@@@
```

The **SPACE$**(*n*) function generates a string consisting of *n* blank spaces.

Changing the Case of Letters

You may want to convert uppercase (capital) letters in a string to lowercase or vice versa. This conversion is useful in a non-case-sensitive search for a particular string pattern in a large file (in other words, "help," "HELP," and "Help" are all be considered the same). These functions are also handy when you are not sure whether a user will input text in uppercase or lowercase letters.

The **UCASE$** and **LCASE$** functions make the following conversions to a string:

- **UCASE$** returns a copy of the string passed to it, with all the lowercase letters converted to uppercase.
- **LCASE$** returns a copy of the string passed to it, with all the uppercase letters converted to lowercase.

Example The following example prints a string unchanged, in all uppercase letters, and in all lowercase letters:

```
Sample$ = "* The ASCII Appendix: a table of useful codes *"
PRINT Sample$
PRINT UCASE$(Sample$)
PRINT LCASE$(Sample$)
```

Output

```
* The ASCII Appendix: a table of useful codes *
* THE ASCII APPENDIX: A TABLE OF USEFUL CODES *
* the ascii appendix: a table of useful codes *
```

Letters that are already uppercase, as well as characters that are not letters, remain unchanged by **UCASE$**; similarly, lowercase letters and characters that are not letters are unaffected by **LCASE$**.

Strings and Numbers

BASIC does not allow a string to be assigned to a numeric variable, nor does it allow a numeric expression to be assigned to a string variable. For example, if you use either of these statements, BASIC will generate the error message `Type mismatch`.

```
TempBuffer$ = 45
Counter% = "45"
```

Instead, use the **STR$** function to return the string representation of a number or the **VAL** function to return the numeric representation of a string:

```
' The following statements are both valid:
TempBuffer$ = STR$(45)
Counter%  = VAL("45")
```

Example Note that **STR$** includes the leading blank space that BASIC prints for positive numbers, as this short example shows:

```
FOR I = 0 TO 9
   PRINT STR$(I);
NEXT
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

If you want to eliminate this space, you can use the **LTRIM$** function, as shown in the following example:

```
FOR I = 0 TO 9
   PRINT LTRIM$(STR$(I));
NEXT
```

Output

```
0123456789
```

Another way to format numeric output is with the **PRINT USING** statement (see the section "Printing Text on the Screen" in Chapter 3, "File and Device I/O").

Changing Strings

The functions mentioned in each of the preceding sections all leave their string arguments unchanged. Changes are made to a copy of the argument.

In contrast, the **MID$** statement (unlike the **MID$** function discussed in the section "Retrieving Characters from Anywhere in a String" earlier in this chapter) changes its argument by embedding another string in it. The embedded string replaces part or all of the original string.

Example The following example replaces parts of a string:

```
Temp$ = "In the sun."
PRINT Temp$

' Replace the "I" with an "O":
MID$(Temp$,1) = "O"

' Replace "sun." with "road":
MID$(Temp$,8) = "road"
PRINT Temp$
```

Output

```
In the sun.
On the road
```

Sample Application: Converting a String to a Number (STRTONUM.BAS)

The following program takes a number that is input as a string, filters any numeric characters (such as commas) out of the string, then converts the string to a number with the **VAL** function.

Functions Used

This program demonstrates the following string-handling functions discussed in this chapter:

- **INSTR**
- **LEN**
- **MID$**
- **VAL**

Program Listing

```
' Input a line:
LINE INPUT "Enter a number with commas: "; A$

' Look only for valid numeric characters (0123456789.-)
' in the input string:
CleanNum$ = Filter$(A$, "0123456789.-")

' Convert the string to a number:
PRINT "The number's value = "; VAL(CleanNum$)
END
' ========================== Filter$ =====================
'         Takes unwanted characters out of a string by
'         comparing them with a filter string containing
'         only acceptable numeric characters
' ========================================================

FUNCTION Filter$ (Txt$, FilterString$) STATIC
   Temp$ = ""
   TxtLength = LEN(Txt$)

   FOR I = 1 TO TxtLength      ' Isolate each character in
      C$ = MID$(Txt$, I, 1)    ' the string.

      ' If the character is in the filter string, save it:
      IF INSTR(FilterString$, C$) <> 0 THEN
         Temp$ = Temp$ + C$
      END IF
   NEXT I

   Filter$ = Temp$
END FUNCTION
```

Chapter 5
Graphics

This chapter shows you how to use the graphics statements and functions of Microsoft BASIC to create a wide variety of shapes, colors, and patterns on your screen. With graphics, you can add a visual dimension to almost any program, whether it's a game, an educational tool, a scientific application, or a financial package.

When you have finished studying this chapter, you will know how to perform the following graphics tasks:

- Use the physical-coordinate system of your personal computer's screen to locate individual pixels, turn those pixels on or off, and change their colors.
- Draw lines.
- Draw and fill simple shapes, such as circles, ovals, and boxes.
- Restrict the area of the screen showing graphics output by using viewports.
- Adjust the coordinates used to locate a pixel by redefining screen coordinates.
- Use color in graphics output.
- Create patterns and use them to fill enclosed figures.
- Copy images and reproduce them in another location on the screen.
- Animate graphics output.

The next section contains important information on what you'll need to run the graphics examples shown in this chapter. Read it first.

Note To learn how to use BASIC's new presentation graphics, see Chapter 6, "Presentation Graphics."

What You Need for Graphics Programs

To run the graphics examples shown in this chapter, your computer must have graphics capability, either built in or in the form of a graphics card such as the Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), or Video Graphics Array (VGA). You also need a video display (either monochrome or color) that supports pixel-based graphics.

Also, these programs all require that your screen be in one of the "screen modes" supporting graphics output. (The screen mode controls the clarity of graphics images, the number of colors available, and the part of the video memory to display.) To select a graphics output mode, use the following statement in your program before using any of the graphics statements or functions described in this chapter:

SCREEN *mode%*

Here, *mode%* can be either 1, 2, 3, 4, 7, 8, 9, 10, 11, 12, or 13, depending on the monitor/adapter combination installed on your computer.

If you are not sure whether or not the users of your programs have hardware that supports graphics, you can use the following simple test:

```
CONST FALSE = 0, TRUE = NOT FALSE

' Test to make sure user has hardware
' with color/graphics capability:
ON ERROR GOTO Message       ' Turn on error trapping.
SCREEN 1                    ' Try graphics mode one.
ON ERROR GOTO 0             ' Turn off error trapping.
IF NoGraphics THEN END      ' End if no graphics hardware.
.
.
.
END

' Error-handling routine:
Message:
   PRINT "This program requires graphics capability."
   NoGraphics = TRUE
   RESUME NEXT
```

If the user has only a monochrome display with no graphics adapter, the **SCREEN** statement produces an error that in turn triggers a branch to the error-handling-routine message. (See Chapter 8, "Error Handling," for more information on error handling.)

Pixels and Screen Coordinates

Shapes and figures on a video display are composed of individual dots of light known as picture elements or "pixels" (or sometimes as "pels") for short. BASIC draws and paints on the screen by turning these pixels on or off and by changing their colors.

A typical screen is composed of a grid of pixels. The exact number of pixels in this grid depends on the hardware you have installed and the screen mode you have selected in the **SCREEN** statement. The larger the number of pixels, the higher the clarity of graphics output. For example, `SCREEN 1` gives a resolution of 320 pixels horizontally by 200 pixels vertically (320 x 200 pixels), while `SCREEN 2` gives a resolution of 640 x 200 pixels. The higher horizontal density in screen mode 2—640 pixels per row versus 320 pixels per row—gives images a sharper, less ragged appearance than they have in screen mode 1.

Depending on the graphics capability of your system, you can use other screen modes that support even higher resolutions (as well as adjust other screen characteristics). Consult online Help for more information.

When your screen is in one of the graphics modes, you can locate each pixel by means of pairs of coordinates. The first number in each coordinate pair tells the number of pixels from the left side of the screen, while the second number in each pair tells the number of pixels from the top of the screen. For example, in screen mode 2 the point in the extreme upper-left corner of the screen has coordinates (0, 0), the point in the center of the screen has coordinates (320, 100), and the point in the extreme lower-right corner of the screen has coordinates (639, 199), as shown in Figure 5.1.

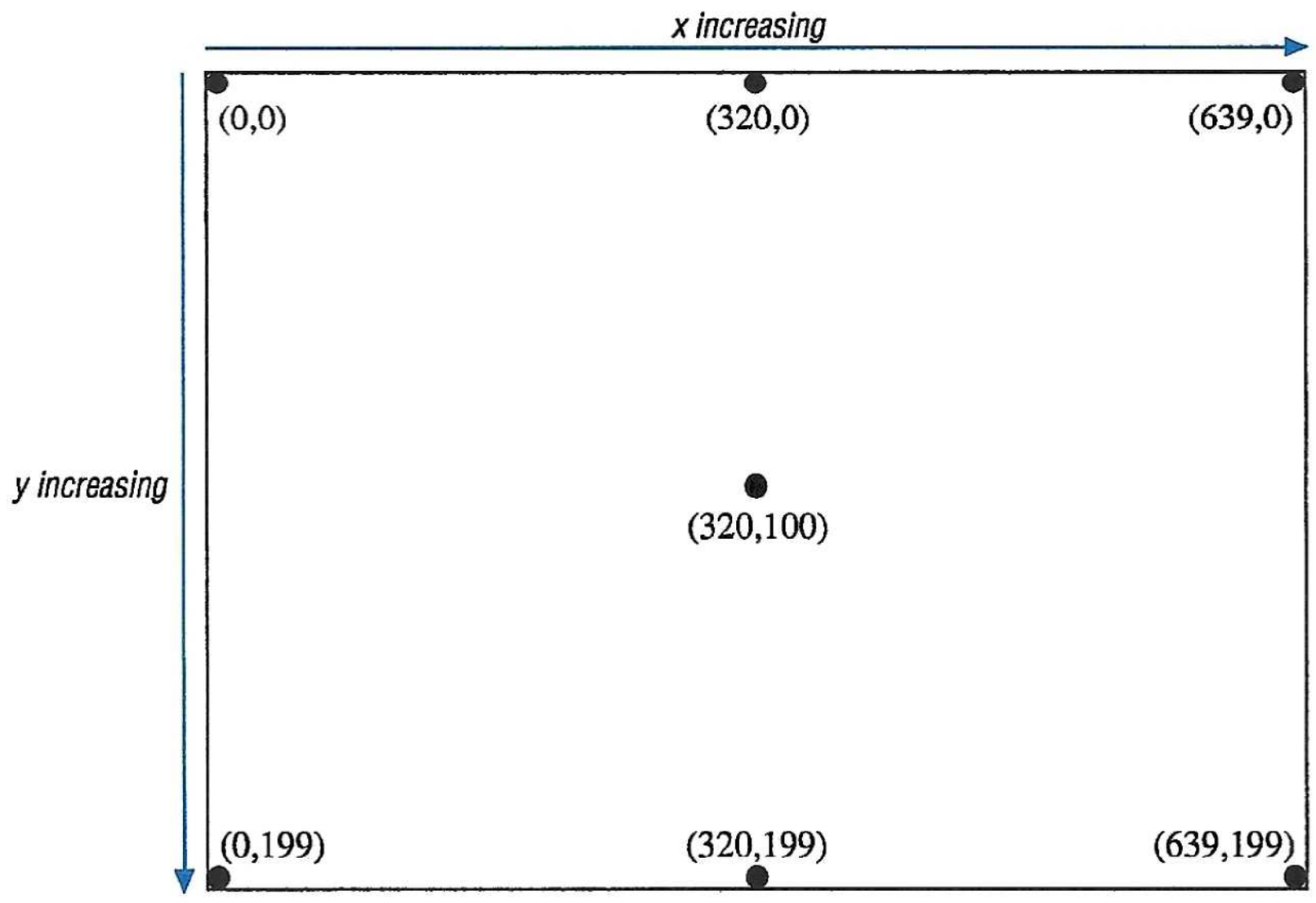


Figure 5.1 Coordinates of Selected Pixels in Screen Mode 2

BASIC uses these screen coordinates to determine where to display graphics (for example, to locate the end points of a line or the center of a circle), as shown in the next section "Drawing Basic Shapes: Points, Lines, Boxes, and Circles."

Graphics coordinates differ from text-mode coordinates specified in a **LOCATE** statement. First, **LOCATE** is not as precise: graphics coordinates pinpoint individual pixels on the screen, whereas coordinates used by **LOCATE** are character positions. Second, text-mode coordinates are given in the form "row, column," as in the following:

```
' Move to the 13th row, 15th column,
' then print the message shown:
LOCATE 13, 15
PRINT "This should appear in the middle of the screen."
```

This is the reverse of graphics coordinates, which are given in the form "column, row." A **LOCATE** statement has no effect on the positioning of graphics output on the screen.

Drawing Basic Shapes: Points, Lines, Boxes, and Circles

You can pass coordinate values to BASIC graphics statements to produce a variety of simple figures, as shown in the following sections.

Plotting Points with PSET and PRESET

The most fundamental level of control over graphics output is simply turning individual pixels on and off. You do this in BASIC with the **PSET** (for pixel set) and **PRESET** (for pixel reset) statements. The statement **PSET** (*x%*, *y%*) gives the pixel at location (*x%*, *y%*) the current foreground color. The **PRESET** (*x%*, *y%*) statement gives the pixel at location (*x%*, *y%*) the current background color.

On monochrome monitors, the foreground color is the color that is used for printed text and is typically white, amber, or light green; the background color on monochrome monitors is typically black or dark green. You can choose another color for **PSET** and **PRESET** to use by adding an optional *color* argument. The syntax is then:

{PSET|PRESET} (*x%*,*y%*),*color%*

See the section "Using Colors" later in this chapter for more information on choosing colors.

Because **PSET** uses the current foreground color by default and **PRESET** uses the current background color by default, **PRESET** without a *color* argument erases a point drawn with **PSET**, as shown in the next example:

```
SCREEN 2  ' 640 x 200 resolution
PRINT "Press any key to end."

DO

   ' Draw a horizontal line from the left to the right:
   FOR X = 0 TO 640
      PSET (X, 100)
   NEXT
```

```
    ' Erase the line from the right to the left:
    FOR X = 640 TO 0 STEP -1
       PRESET (X, 100)
    NEXT

LOOP UNTIL INKEY$ <> ""
END
```

While it is possible to draw any figure using only **PSET** statements to manipulate individual pixels, the output tends to be rather slow, since most pictures consist of many pixels. BASIC has several statements that dramatically increase the speed with which simple figures—such as lines, boxes, and ellipses—are drawn, as shown in the following sections "Drawing Lines and Boxes with LINE" and "Drawing Circles and Ellipses with CIRCLE."

Drawing Lines and Boxes with LINE

When using **PSET** or **PRESET**, you specify only one coordinate pair since you are dealing with only one point on the screen. With **LINE**, you specify two pairs, one for each end of a line segment. The simplest form of the **LINE** statement is as follows:

LINE(*x1%*,*y1%*)–(*x2%*,*y2%*)

where (*x1%*, *y1%*) are the coordinates of one end of a line segment and (*x2%*, *y2%*) are the coordinates of the other. For example, the following statement draws a straight line from the pixel with coordinates (10, 10) to the pixel with coordinates (150, 200):

```
SCREEN 1
LINE (10, 10)-(150, 200)
```

Note that BASIC does not care about the order of the coordinate pairs: a line from (*x1%*, *y1%*) to (*x2%*, *y2%*) is the same as a line from (*x2%*, *y2%*) to (*x1%*, *y1%*). This means you could also write the preceding statement as:

```
SCREEN 1
LINE (150, 200)-(10, 10)
```

However, reversing the order of the coordinates could have an effect on graphics statements that follow, as shown in the next section.

Using the STEP Keyword

Up to this point, screen coordinates have been presented as absolute values measuring the horizontal and vertical distances from the extreme upper-left corner of the screen, which has coordinates (0, 0). However, by using the **STEP** keyword in any of the following graphics statements, you can make the coordinates that follow **STEP** relative to the last point referred to on the screen:

CIRCLE
GET
LINE
PAINT
PRESET
PSET
PUT

If you picture images as being drawn on the screen by a tiny paintbrush exactly the size of one pixel, then the last point referenced is the location of this paintbrush, or "graphics cursor," when it finishes drawing an image. For example, the following statements leave the graphics cursor at the pixel with coordinates (100, 150):

```
SCREEN 2
LINE (10, 10)-(100, 150)
```

If the next graphics statement in the program is as follows, then the point plotted by **PSET** is not in the upper-left quadrant of the screen:

```
PSET STEP(20, 20)
```

Instead, **STEP** has made the coordinates (20, 20) relative to the last point referenced, which has coordinates (100, 150). This makes the absolute coordinates of the point (100 + 20, 150 + 20) or (120, 170).

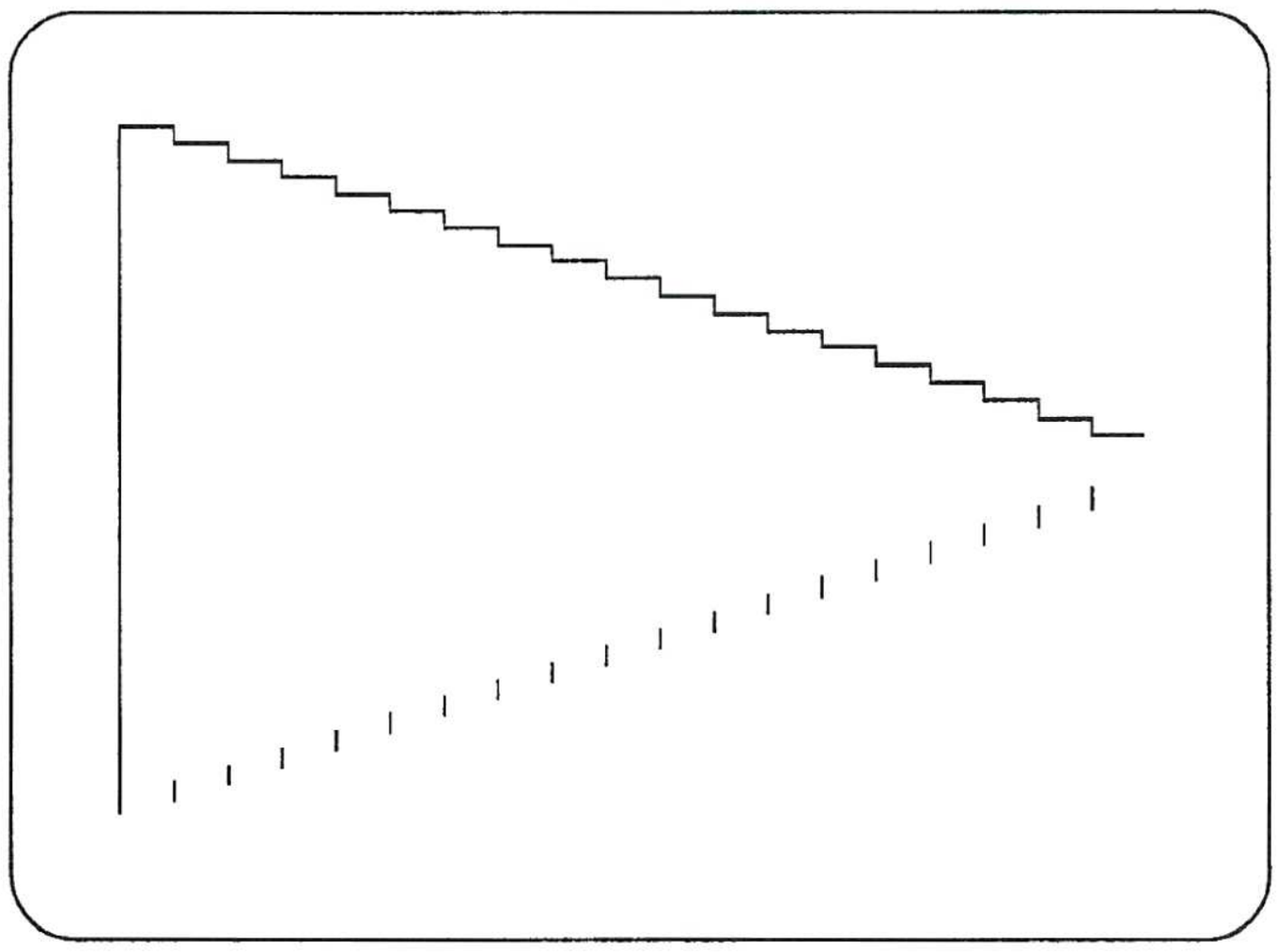
Example

In the preceding example, the last point referenced is determined by a preceding graphics statement. You can also establish a reference point within the same statement, as shown in this example:

```
' Set 640 x 200 pixel resolution, and make the last
' point referenced the center of the screen:
SCREEN 2

' Draw a line from the lower-left corner of the screen
' to the upper-left corner:
LINE STEP(-310, 100) -STEP(0, -200)

' Draw the "stair steps" down from the upper-left corner
' to the right side of the screen:
FOR I% = 1 TO 20
   LINE -STEP(30, 0)' Draw the horizontal line.
   LINE -STEP(0, 5) ' Draw the vertical line.
NEXT
```

```
' Draw the unconnected vertical line segments from the
' right side to the lower-left corner:
FOR I% = 1 TO 20
   LINE STEP(-30, 0) -STEP(0, 5)
NEXT

SLEEP ' Wait for a keystroke.
```

Output



Note

Note the **SLEEP** statement at the end of the last program. If you are running a compiled, stand-alone BASIC program that produces graphics output, your program needs a mechanism like this at the end to hold the output on the screen. Otherwise, it vanishes from the screen before the user notices it.

Drawing Boxes

Using the forms of the **LINE** statement already presented, it is quite easy to write a short program that connects four straight lines to form a box, as shown here:

```
SCREEN 1  ' 320 x 200 pixel resolution
```

```
' Draw a box measuring 120 pixels on a side:
LINE (50, 50)-(170, 50)
LINE -STEP(0, 120)
LINE -STEP(-120, 0)
LINE -STEP(0, -120)
```

However, BASIC provides an even simpler way to draw a box, using a single **LINE** statement with the **B** (for box) option. The **B** option is shown in the next example, which produces the same output as the four **LINE** statements in the preceding program:

```
SCREEN 1  ' 320 x 200 pixel resolution

' Draw a box with coordinates (50, 50) for the upper-left
' corner, and (170, 170) for the lower-right corner:
LINE (50, 50)-(170, 170), , B
```

When you add the **B** option, the **LINE** statement no longer connects the two points you specify with a straight line; instead, it draws a rectangle whose opposite corners (upper left and lower right) are at the locations specified.

Two commas precede the **B** in the last example. The first comma functions here as a placeholder for the unused *color&* argument, which allows you to pick the color for a line or the sides of a box. (See the section "Using Colors" later in this chapter for more information on the use of color.)

As before, it does not matter what order the coordinate pairs are given in, so the rectangle from the last example could also be drawn with this statement:

```
LINE (170, 170)-(50, 50), , B
```

Adding the F (for fill) option after **B** fills the interior of the box with the same color used to draw the sides. With a monochrome display, this is the same as the foreground color used for printed text. If your hardware has color capabilities, you can change this color with the optional *color&* argument (see the section "Selecting a Color for Graphics Output" later in this chapter).

The syntax introduced here for drawing a box is the general syntax used in BASIC to define a rectangular graphics region, and it also appears in the **GET** and **VIEW** statements:

{**GET**|**LINE**|**VIEW**} (*x1!*,*y1!*)–(*x2!*,*y2!*), ...

Here, (*x1!*, *y1!*) and (*x2!*, *y2!*) are the coordinates of diagonally opposite corners of the rectangle (upper left and lower right). (See "Defining a Graphics Viewport" later in this chapter for a discussion of **VIEW**, and "Basic Animation Techniques" later in this chapter for information on **GET** and **PUT**.)

Drawing Dotted Lines

The previous sections explain how to use **LINE** to draw solid lines and use them in rectangles; that is, no pixels are skipped. Using yet another option with **LINE**, you can draw dashed or dotted lines instead. This process is known as "line styling." The following is the syntax for drawing a single dashed line from point (*x1*, *y1*) to point (*x2*, *y2*) using the current foreground color:

LINE (*x1!*,*y1!*)–(*x2!*,*y2!*),,[[**B**]],*style%*

Here *style%* is a 16-bit decimal or hexadecimal integer. The **LINE** statement uses the binary representation of the line-style argument to create dashes and blank spaces, with a 1 bit meaning "turn on the pixel," and a 0 bit meaning "leave the pixel off." For example, the hexadecimal integer &HCCCC is equal to the binary integer 1100110011001100, and when used as a *style%* argument it draws a line alternating two pixels on, two pixels off.

Example The following example shows different dashed lines produced using different values for *style%*:

```
SCREEN 2  ' 640 x 200 pixel resolution

' Style data:
DATA &HCCCC, &HFF00, &HF0F0
DATA &HF000, &H7777, &H8888

Row% = 4
Column% = 4
XLeft% = 60
XRight% = 600
Y% = 28

FOR I% = 1 TO 6
   READ Style%
   LOCATE Row%, Column%
   PRINT HEX$(Style%)
   LINE (XLeft%, Y%)-(XRight%,Y%), , , Style%
   Row% = Row% + 3
   Y% = Y% + 24
NEXT
```

Output

```
CCCC ..........................................................
FF00 - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
F0F0 ----------------------------------------------------------
F000 - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
7777 ..........................................................
8888 ..........................................................
```

Drawing Circles and Ellipses with CIRCLE

The **CIRCLE** statement draws a variety of circular and elliptical, or oval, shapes. In addition, **CIRCLE** draws arcs (segments of circles) and pie-shaped wedges. In graphics mode you can produce just about any kind of curved line with some variation of **CIRCLE.**

Drawing Circles

To draw a circle, you need to know only two things: the location of its center and the length of its radius (the distance from the center to any point on the circle). With this information and a reasonably steady hand (or better yet, a compass), you can produce an attractive circle.

Similarly, BASIC needs only the location of a circle's center and the length of its radius to draw a circle. The simplest form of the **CIRCLE** syntax is:

CIRCLE [[STEP]] (*x!*,*y!*),*radius!*

In this statement, *x!*, *y!* are the coordinates of the center, and *radius!* is the radius of the circle. The next lines of code draw a circle with center (200, 100) and radius 75:

```
SCREEN 2
CIRCLE (200, 100), 75
```

You could rewrite the preceding example as follows, making the same circle but using **STEP** to make the coordinates relative to the center rather than to the upper-left corner:

```
SCREEN 2          ' Uses center of screen (320,100) as the
                  ' reference point for the CIRCLE statement:
CIRCLE STEP(-120, 0), 75
```

Drawing Ellipses

The **CIRCLE** statement automatically adjusts the "aspect ratio" to make sure that circles appear round and not flattened on your screen. However, you may need to adjust the aspect ratio to make circles come out right on your monitor, or you may want to change the aspect ratio to draw the oval figure known as an ellipse. In either case, use this syntax:

CIRCLE[**STEP**] (*x!*,*y!*),*radius!*,,,,*aspect!*

Here, *aspect!* is a positive real number. (See "Drawing Shapes to Proportion with the Aspect Ratio" later in this chapter for more information on the aspect ratio and how to calculate it for different screen modes.)

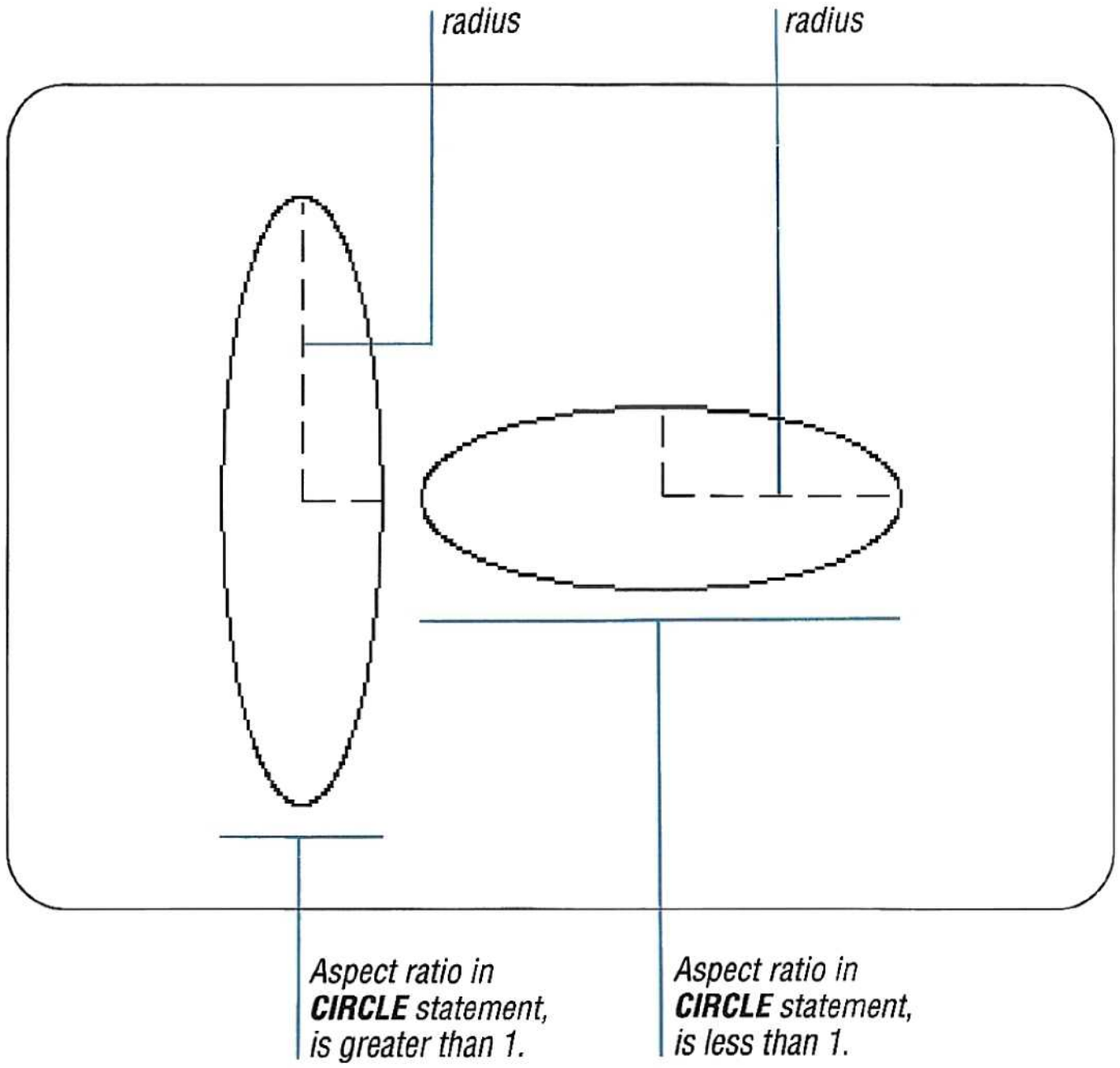
The extra commas between *radius!* and *aspect!* are placeholders for other options that tell **CIRCLE** which color to use (if you have a color monitor/adapter and are using one of the screen modes that support color), or whether to draw an arc or wedge. (See the sections "Drawing Arcs" and "Selecting a Color for Graphics Output" later in this chapter for more information on these options.)

Since the argument *aspect!* specifies the ratio of the vertical to horizontal dimensions, large values for *aspect!* produce ellipses stretched out along the vertical axis, while small values for *aspect!* produce ellipses stretched out along the horizontal axis. Since an ellipse has two radii—one horizontal x-radius and one vertical y-radius—BASIC uses the single *radius!* argument in a **CIRCLE** statement as follows: if *aspect!* is less than one, then *radius!* is the x-radius; if *aspect!* is greater than or equal to one, then *radius!* is the y-radius.

Example The following example and its output show how different *aspect!* values affect whether the **CIRCLE** statement uses the *radius!* argument as the x-radius or the y-radius of an ellipse:

```
SCREEN 1

' This draws the ellipse on the left:
CIRCLE (60, 100), 80, , , , 3

' This draws the ellipse on the right:
CIRCLE (180, 100), 80, , , , 3/10
```

Output



Drawing Arcs

An arc is a segment of a ellipse, in other words a short, curved line. To understand how the **CIRCLE** statement draws arcs, you need to know how BASIC measures angles.

BASIC uses the radian as its unit of angle measure, not only in the **CIRCLE** statement, but also in the intrinsic trigonometric functions such as **COS**, **SIN**, or **TAN**. (The one exception to this use of radians is the **DRAW** statement, which expects angle measurements in degrees. See "DRAW: a Graphics Macro Language" later in this chapter for more information about **DRAW**.)

The radian is closely related to the radius of a circle. In fact, the word "radian" is derived from the word "radius." The circumference of a circle equals $2 * \pi * radius$, where π is equal to approximately 3.14159265. Similarly, the number of radians in one complete angle of revolution (or 360) equals $2 * \pi$, or a little more than 6.28.

If you are more used to thinking of angles in terms of degrees, here are some common equivalences:

Angle in degrees	Angle in radians
360	2π (approximately 6.283)
180	π (approximately 3.142)
90	$\pi/2$ (approximately 1.571)
60	$\pi/3$ (approximately 1.047)

If you picture a clock face on the screen, **CIRCLE** measures angles by starting at the three o'clock position and rotating counterclockwise, as shown in Figure 5.2:

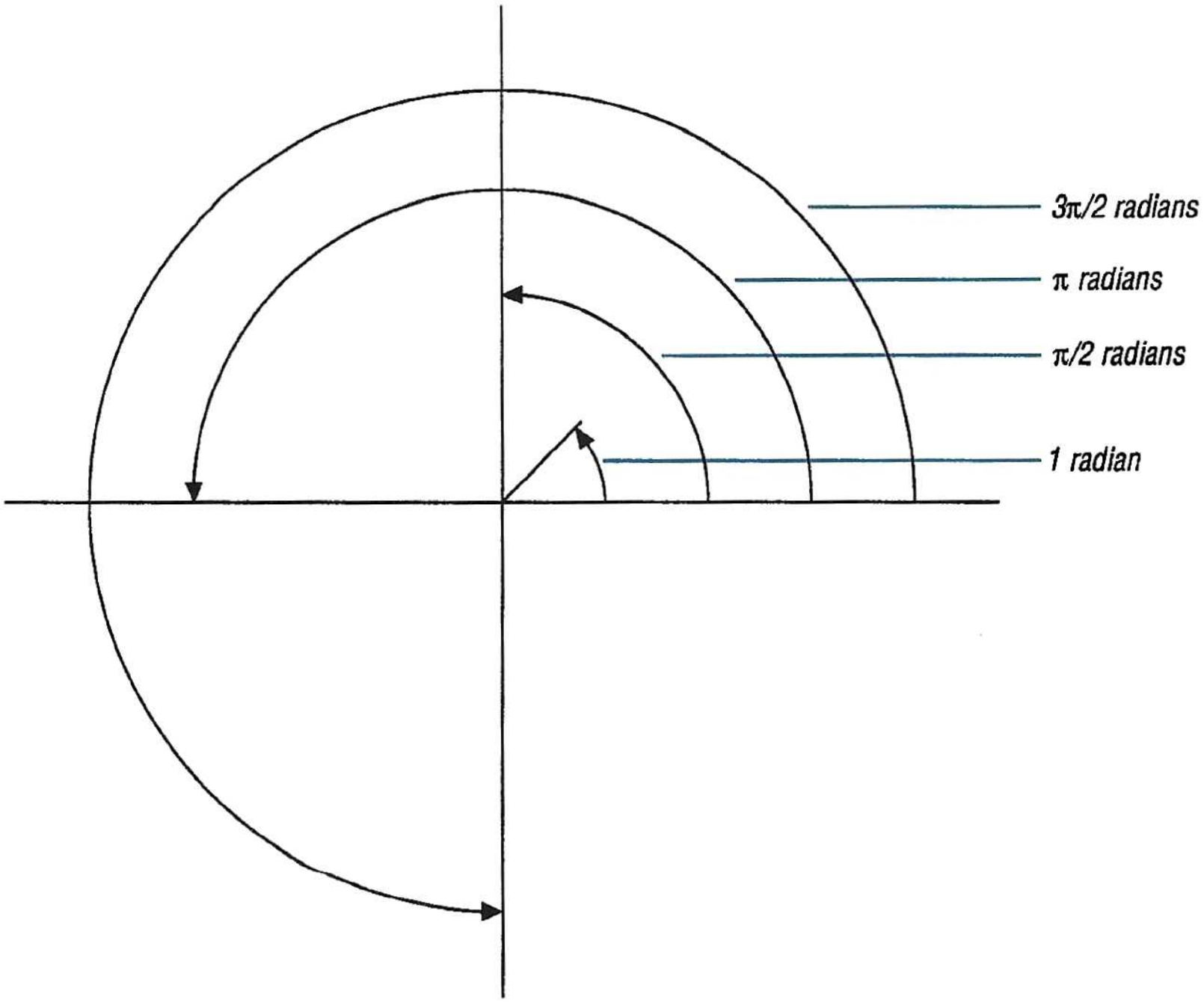


Figure 5.2 How Angles Are Measured for CIRCLE

The general formula for converting from degrees to radians is to multiply degrees by π/180.

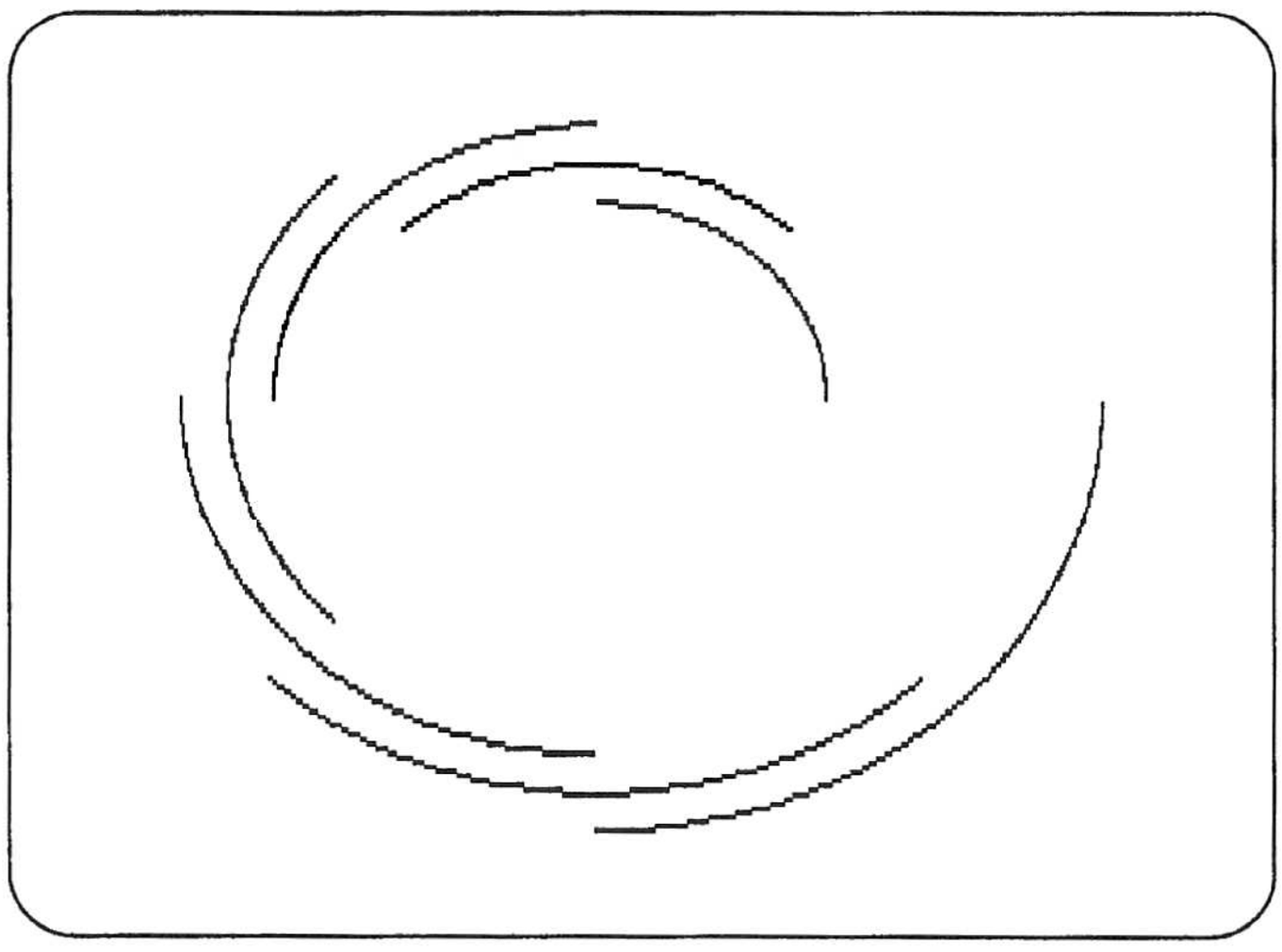
To draw an arc, give angle arguments defining the arc's limits:

CIRCLE [[**STEP**]](*x!*,*y!*),*radius!*, [[*color&*]],*start!*,*end!* [[,*aspect!*]]

Example The **CIRCLE** statements in the next example draw seven arcs, with the innermost arc starting at the three o'clock position (0 radians) and the outermost arc starting at the six o'clock position (3π/2 radians), as you can see from the output:

```
SCREEN 2
CLS

CONST PI = 3.141592653589#      ' Double-precision constant

StartAngle = 0
FOR Radius% = 100 TO 220 STEP 20
   EndAngle = StartAngle + (PI / 2.01)
   CIRCLE (320, 100), Radius%, , StartAngle, EndAngle
   StartAngle = StartAngle + (PI / 4)
NEXT Radius%
```

Output



Drawing Pie Shapes

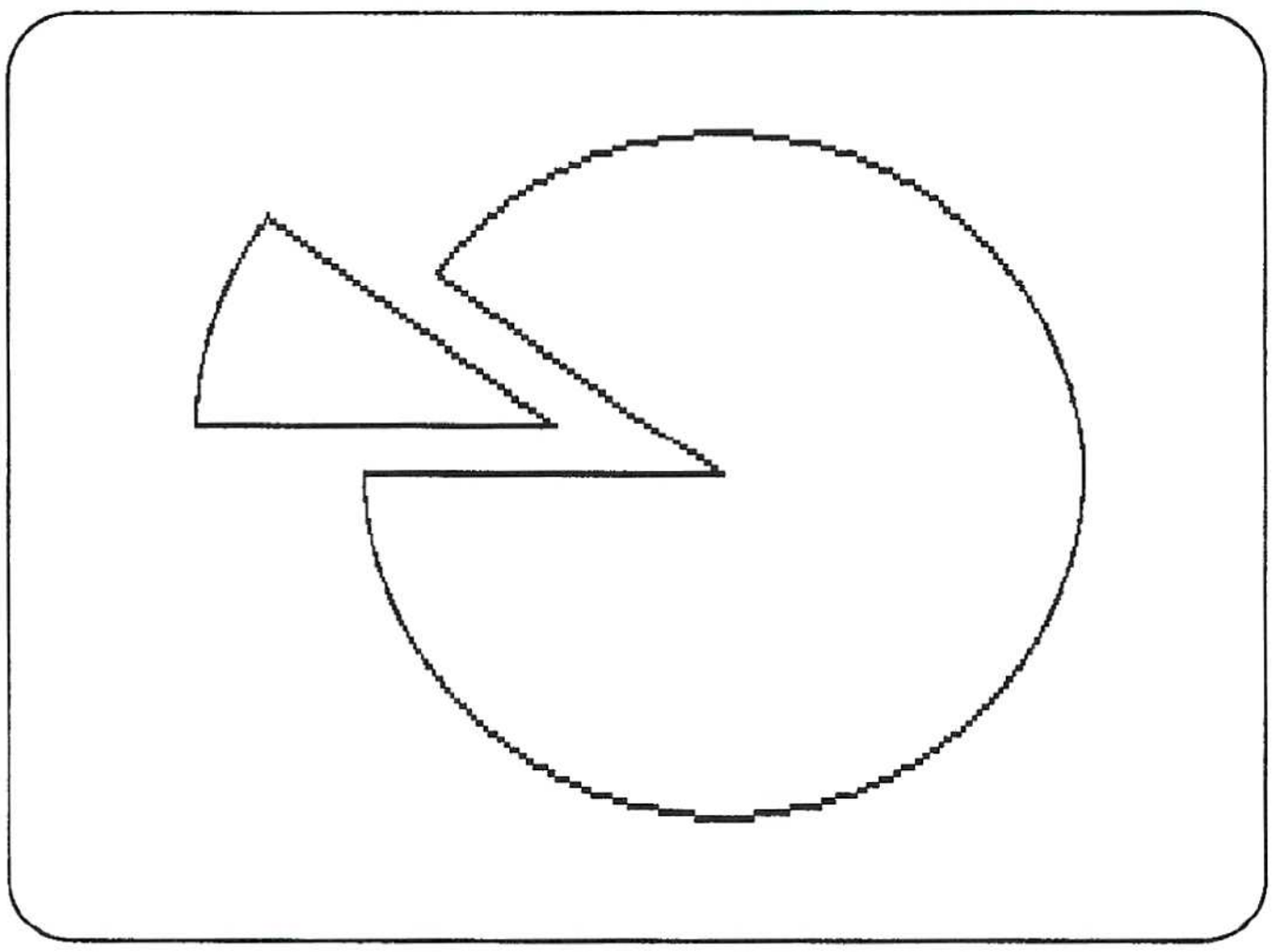
By making either of **CIRCLE**'s *start!* or *end!* arguments negative, you can connect the arc at its beginning or ending point with the center of the circle. By making both arguments negative, you can draw shapes ranging from a wedge that resembles a slice of pie to the pie itself with the piece missing.

Example

This example code draws a pie shape with a piece missing:

```
SCREEN 2

CONST RADIUS = 150, PI = 3.1415926535898#

StartAngle = 2.5
EndAngle = PI

' Draw the wedge:
CIRCLE (320, 100), RADIUS, , -StartAngle, -EndAngle

' Swap the values for the start and end angles:
SWAP StartAngle, EndAngle

' Move the center 10 pixels down and 70 pixels to the
' right, then draw the "pie" with the wedge missing:
CIRCLE STEP(70, 10), RADIUS, , -StartAngle, -EndAngle
```

Output



Drawing Shapes to Proportion with the Aspect Ratio

As discussed earlier in "Drawing Ellipses," BASIC's CIRCLE statement automatically corrects the aspect ratio, which determines how figures are scaled on the screen. However, with other graphics statements you need to scale horizontal and vertical dimensions yourself to make shapes appear with correct proportions. For example, although the following statement draws a rectangle that measures 100 pixels on all sides, it does not look like a square:

```
SCREEN 1
LINE (0, 0)-(100, 100), , B
```

In fact, this is not an optical illusion; the rectangle really is taller than it is wide. This is because in screen mode 1 there is more space between pixels vertically than horizontally. To draw a perfect square, you have to change the aspect ratio.

The aspect ratio is defined as follows: in a given screen mode consider two lines, one vertical and one horizontal, that appear to have the same length. The aspect ratio is the number of pixels in the vertical line divided by the number of pixels in the horizontal line. This ratio depends on two factors:

- Because of the way pixels are spaced on most screens, a horizontal row has more pixels than a vertical column of the exact same physical length in all screen modes except modes 11 and 12.
- The standard personal computer's video-display screen is wider than it is high. Typically, the ratio of screen width to screen height is 4:3.

To see how these two factors interact to produce the aspect ratio, consider a screen after a `SCREEN 1` statement, which gives a resolution of 320 x 200 pixels. If you draw a rectangle from the top of the screen to the bottom, and from the left side of the screen three-fourths of the way across, you have a square, as shown in Figure 5.3.

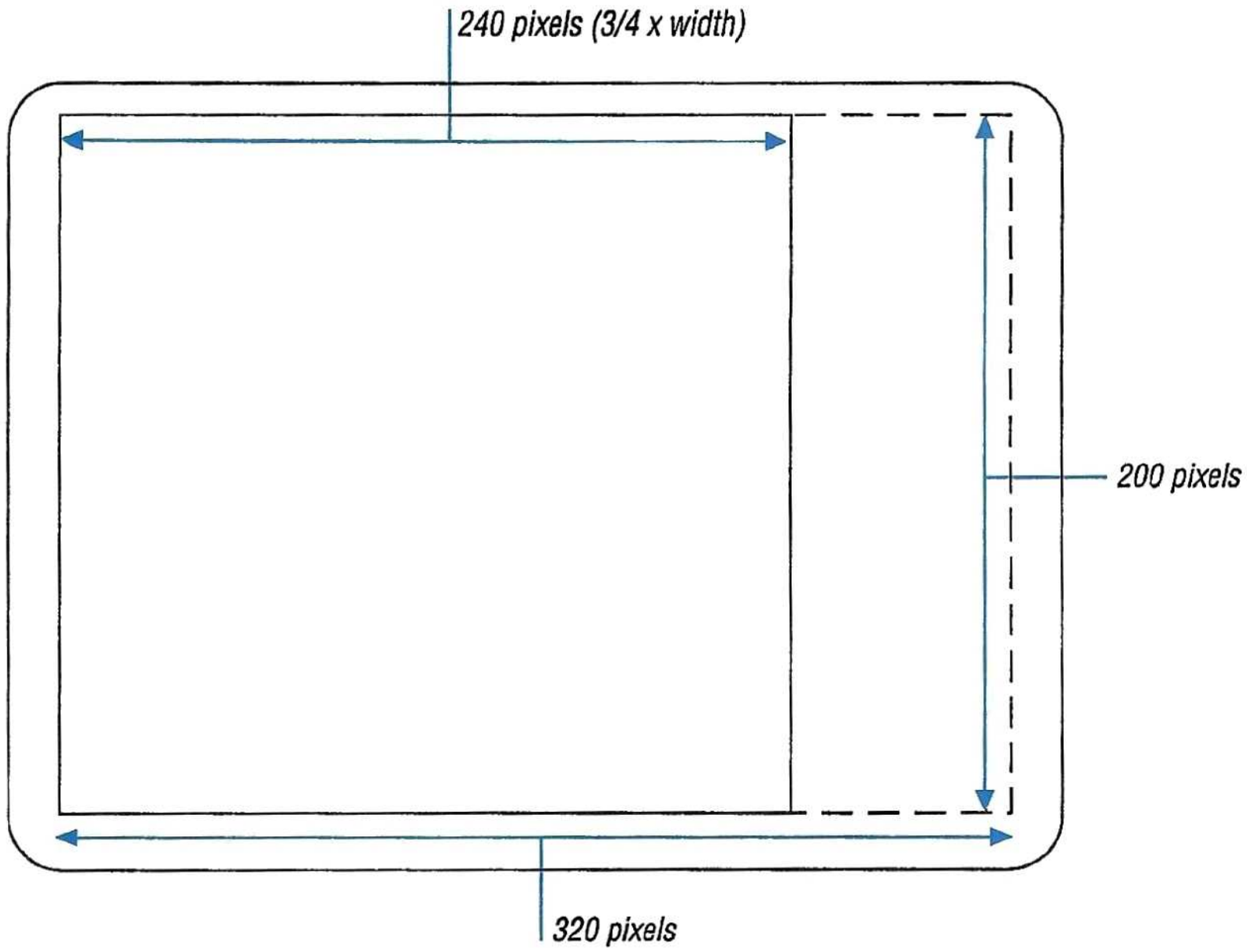


Figure 5.3 The Aspect Ratio in Screen Mode 1

As you can see from the diagram, this square has a height of 200 pixels and a width of 240 pixels. The ratio of the square's height to its width (200 / 240 or, when simplified, 5 / 6) is the aspect ratio for this screen resolution. In other words, to draw a square in 320 x 200 resolution, make its height in pixels equal to 5 / 6 times its width in pixels, as shown in the next example:

```
SCREEN 1  ' 320 x 200 pixel resolution

' The height of this box is 100 pixels, and the width is
' 120 pixels, which makes the ratio of the height to the
' width equal to 100/120, or 5/6. The result is a square:
LINE (50, 50) -STEP(120, 100), , B
```

The formula for calculating the aspect ratio for a given screen mode is:

(4 / 3) * (*ypixels* / *xpixels*)

In this formula, *xpixels* by *ypixels* is the current screen resolution. In screen mode 1, this formula shows the aspect ratio to be (4 / 3) * (200 / 320), or 5 / 6; in screen mode 2, the aspect ratio is (4 / 3) * (200 / 640), or 5 / 12.

If you have a video display with a ratio of width to height that is not equal to 4:3, use the more general form of the formula for computing the aspect ratio:

(*screenwidth* / *screenheight*) * (*ypixels* / *xpixels*)

The **CIRCLE** statement can be made to draw an ellipse by varying the value of the *aspect* argument, as shown in "Drawing Ellipses" earlier in this chapter.

Defining a Graphics Viewport

The graphics examples presented so far have all used the entire video-display screen as their drawing board, with absolute coordinate distances measured from the extreme upper-left corner of the screen.

However, using the **VIEW** statement you can also define a kind of miniature screen (known as a "graphics viewport") inside the boundaries of the physical screen. Once it is defined, all graphics operations take place within this viewport. Any graphics output outside the viewport boundaries is "clipped"; that is, any attempt to plot a point outside the viewport is ignored. There are two main advantages to using a viewport:

- A viewport makes it easy to change the size and position of the screen area where graphics appear.
- Using **CLS 1**, you can clear the screen inside a viewport without disturbing the screen outside the viewport.

Note

Refer to Chapter 3, "File and Device I/O," to learn how to create a "text viewport" for output printed on the screen.

The general syntax for **VIEW** is as follows:

VIEW [[**SCREEN**]] (*x1!*,*y1!*)–(*x2!*,*y2!*) [[, [[*color&*]] , [[*border&*]]]]

The coordinates (*x1*, *y1*) and (*x2*, *y2*) define the corners of the viewport, using the standard BASIC syntax for rectangles (see the section "Drawing Boxes" earlier in this chapter). Note that the **STEP** option is not allowed with **VIEW**. The optional *color&* and *border&* arguments allow you to choose a color for the interior and edges, respectively, of the viewport rectangle. See the section "Using Colors" later in this chapter for more information on setting and changing colors.

The **VIEW** statement without arguments makes the entire screen the viewport. Without the **SCREEN** option, the **VIEW** statement makes all pixel coordinates relative to the viewport, rather than the full screen. In other words, after the following **VIEW** statement, the pixel plotted with the **PSET** statement is visible, since it is 10 pixels below and 10 pixels to the right of the upper-left corner of the viewport:

```
VIEW (50, 60)-(150, 175)
PSET (10, 10)
```

Note that this makes the pixel's absolute screen coordinates (50 + 10, 60 + 10) or (60, 70).

In contrast, the **VIEW** statement with the **SCREEN** option keeps all coordinates absolute; that is, coordinates measure distances from the side of the screen, not from the sides of the viewport. Therefore, after the following **VIEW SCREEN** statement the pixel plotted with the **PSET** is not visible, since it is 10 pixels below and 10 pixels to the right of the upper-left corner of the screen — outside the viewport:

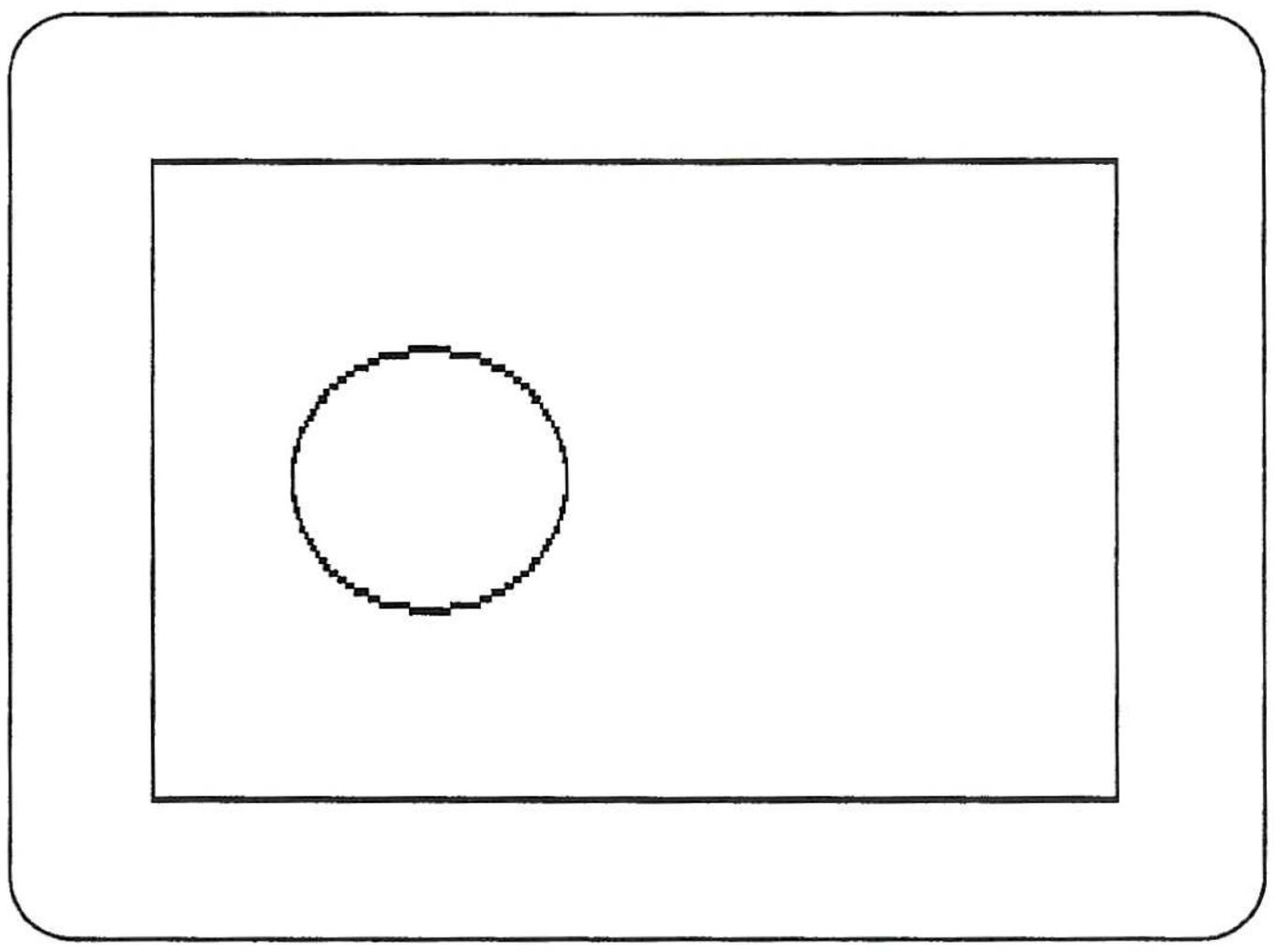
```
VIEW SCREEN (50, 60)-(150, 175)
PSET (10, 10)
```

Examples

The output from the next two examples should further clarify the distinction between **VIEW** and **VIEW SCREEN**:

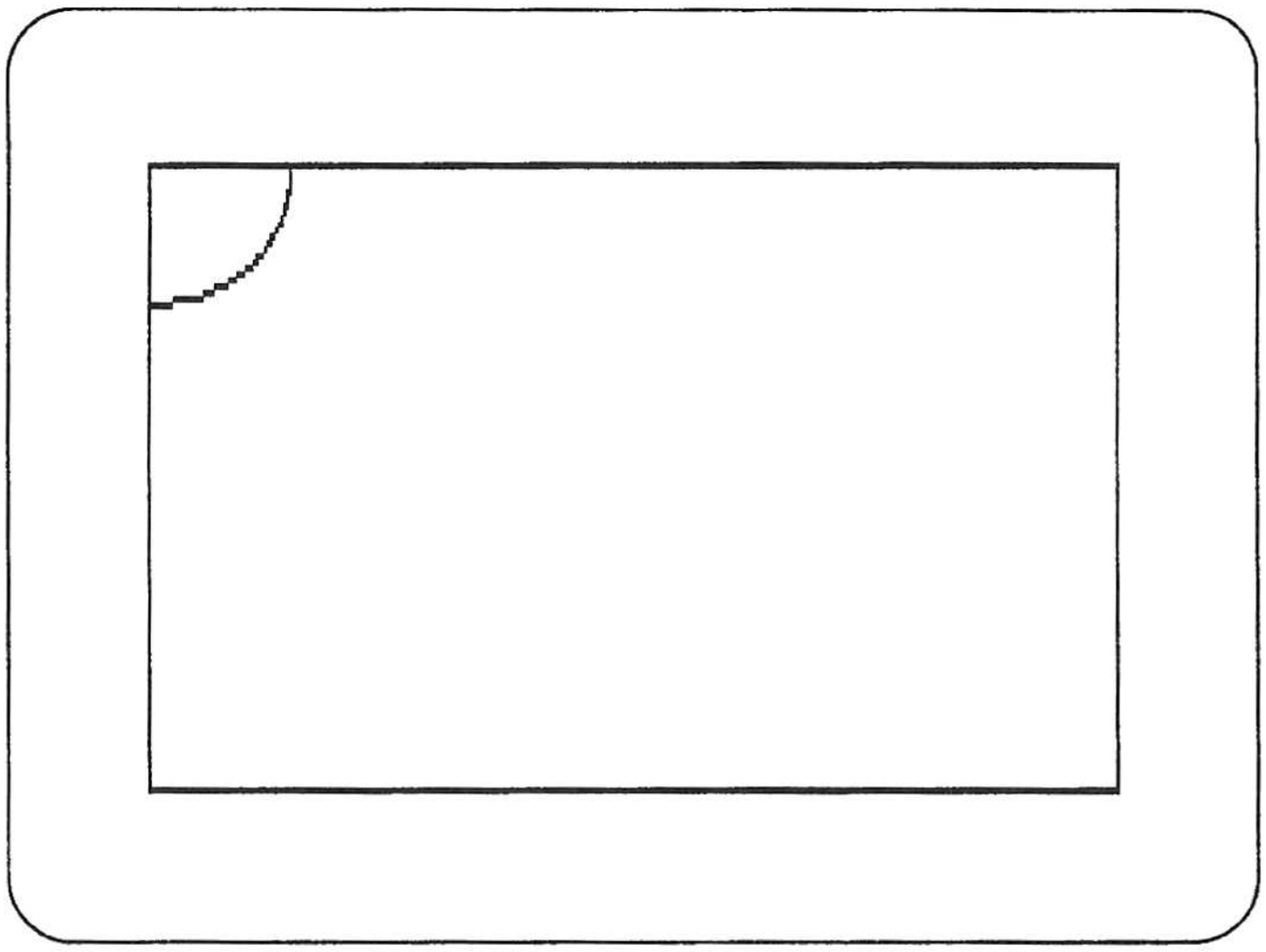
```
SCREEN 2
VIEW (100, 50)-(450, 150), , 1

' This circle's center point has absolute coordinates
' (100 + 100, 50 + 50), or (200, 100):
CIRCLE (100, 50), 50
```

Output Using VIEW



```
SCREEN 2

' This circle's center point has absolute coordinates (100, 50),
' so only part of the circle appears within the viewport:
VIEW SCREEN (100, 50)-(450, 150), , 1
CIRCLE (100, 50), 50
```

Output Using VIEW SCREEN



Note that graphics output located outside the current viewport is clipped by the viewport's edges and does not appear on the screen.

Redefining Viewport Coordinates with WINDOW

This section shows you how to use the **WINDOW** statement and your own coordinate system to redefine pixel coordinates.

In the preceding sections, the coordinates used to locate pixels on the screen represent actual physical distances from the upper-left corner of the screen (or the upper-left corner of the current viewport, if it has been defined with a **VIEW** statement). These are known as "physical coordinates." The "origin," or reference point, for physical coordinates is always the upper-left corner of the screen or viewport, which has coordinates (0, 0).

As you move down the screen and to the right, *x* values (horizontal coordinates) and *y* values (vertical coordinates) get bigger, as shown in the upper diagram of Figure 5.4. While this scheme is standard for video displays, it may seem counterintuitive if you have used other coordinate systems to draw graphs. For example, on the Cartesian grid used in mathematics, *y* values get bigger toward the top of a graph and smaller toward the bottom.

With BASIC's **WINDOW** statement, you can change the way pixels are addressed to use any coordinate system you choose, thus freeing you from the limitations of using physical coordinates.

The general syntax for **WINDOW** is as follows:

WINDOW [[**SCREEN**] (*x1!*,*y1!*)–(*x2!*,*y2!*)]

The coordinates *y1!*, *y2!*, *x1!*, and *x2!* are real numbers specifying the top, bottom, left, and right sides of the window, respectively. These numbers are known as "window coordinates." For example, the following statement remaps your screen so that it is bounded on the top and bottom by the lines `y = 10` and `y = -15` and on the left and right by the lines `x = -25` and `x = 5`:

```
WINDOW (-25, -15)-(5, 10)
```

After a **WINDOW** statement, *y* values get bigger toward the top of the screen. In contrast, after a **WINDOW SCREEN** statement, *y* values get bigger toward the bottom of the screen. Figure 5.4 shows the effects of a **WINDOW** statement and a **WINDOW SCREEN** statement on a line drawn in screen mode 2. Note also how both of these statements change the coordinates of the screen corners. A **WINDOW** statement with no arguments restores the regular physical coordinate system.

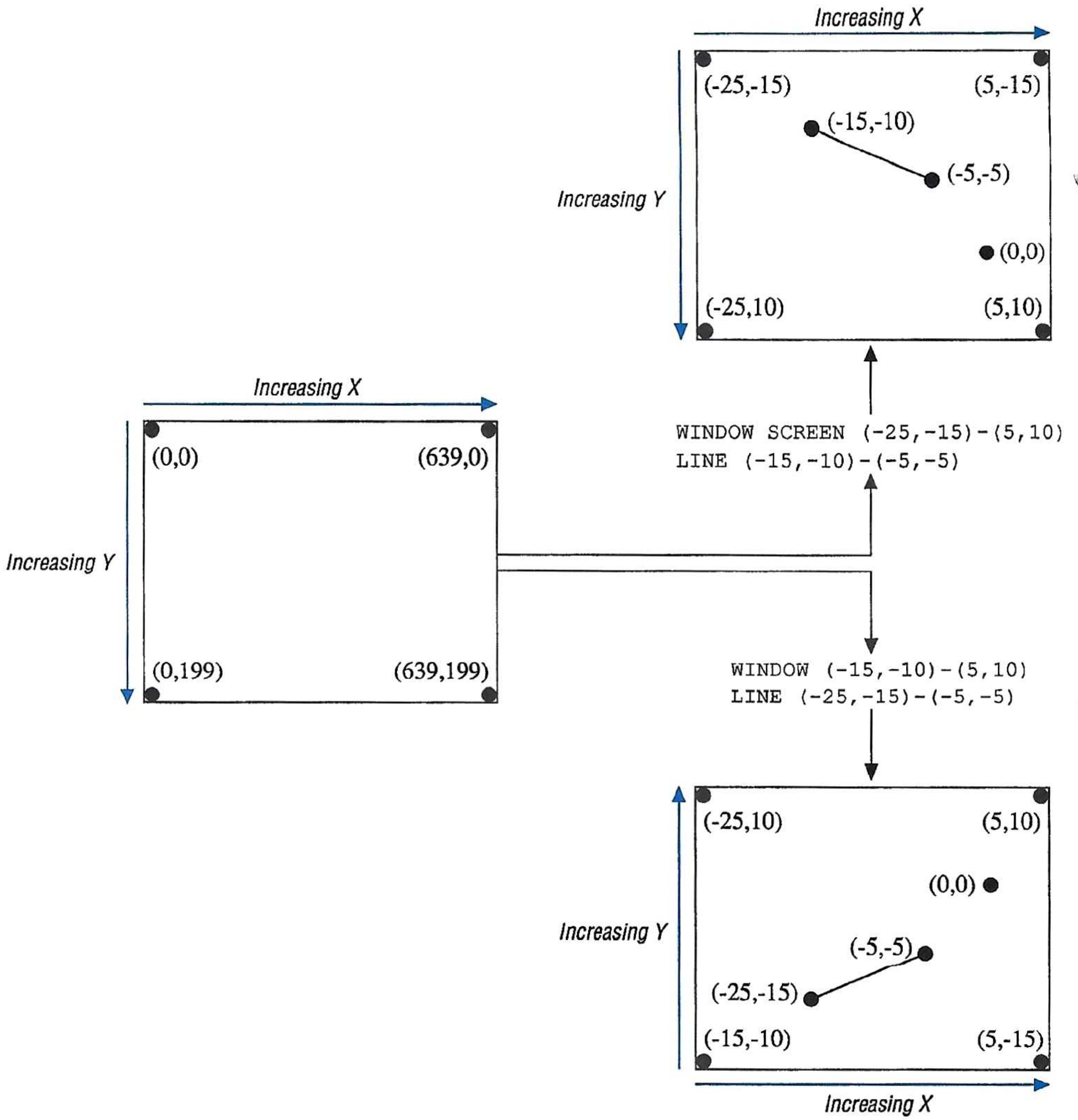
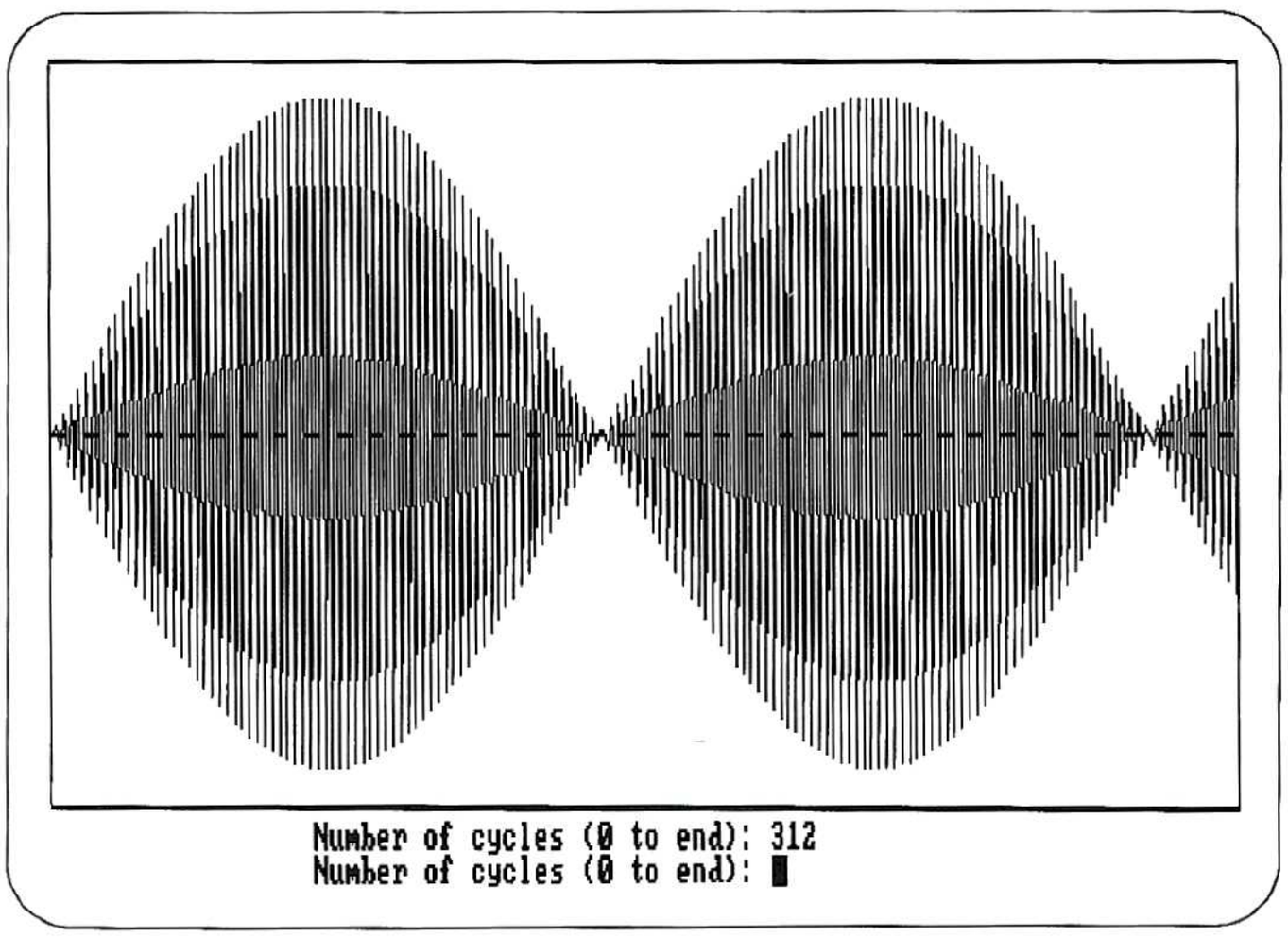


Figure 5.4 WINDOW Contrasted with WINDOW SCREEN

Example The following example uses **VIEW** and **WINDOW** to simplify writing a program to graph the sine-wave function for angle values from 0 radians to π radians (or 0° to 180°). This program is in the file named SINEWAVE.BAS on the Microsoft BASIC distribution disks.

```
SCREEN 2

' Viewport sized to proper scale for graph:
VIEW (20, 2)-(620, 172), , 1
CONST PI = 3.141592653589#

' Make window large enough to graph sine wave from
' 0 radians to pi radians:
WINDOW (0, -1.1)-(2 * PI, 1.1)
Style% = &HFF00          ' Use to make dashed line.
VIEW PRINT 23 TO 24      ' Scroll printed output in rows 23, 24.
DO
   PRINT TAB(20);
   INPUT "Number of cycles (0 to end): ", Cycles
   CLS
   LINE (2 * PI, 0)-(0, 0), , , Style%  ' Draw the x-axis.
   IF Cycles > 0 THEN

      '  Start at (0,0) and plot the graph:
      FOR X = 0 TO 2 * PI STEP .01
    Y = SIN(Cycles * X) ' Calculate the y-coordinate.
    LINE -(X, Y)        ' Draw a line to new point.
      NEXT X
   END IF
LOOP WHILE Cycles > 0
```

Output



The Order of Coordinate Pairs

As with the other BASIC graphics statements that define rectangular areas (**GET**, **LINE**, and **VIEW**), the order of coordinate pairs in a **WINDOW** statement is unimportant. In the following example, the first pair of statements has the same effect as the second pair of statements:

```
VIEW (100, 20)-(300, 120)
WINDOW (-4, -3)-(0, 0)

VIEW (300, 120)-(100, 20)
WINDOW (0, 0)-(-4, -3)
```

Keeping Track of Window and Physical Coordinates

The **PMAP** and **POINT** functions are useful for keeping track of physical and view coordinates. **POINT**(*number%*) tells you the current location of the graphics cursor by returning either the physical or view coordinates (depending on the value for *number%*) of the last point referenced in a graphics statement. **PMAP** allows you to translate physical coordinates to view coordinates and vice versa. The physical coordinate values returned by **PMAP** are always relative to the current viewport.

Examples

The following example shows the different values that are returned by **POINT** (*number%*) for *number%* values of 0, 1, 2, or 3:

```
SCREEN 2
' Define the window:
WINDOW (-10, -30)-(-5, -10)
' Draw a line from the point with window coordinates (-9,-28)
' to the point with window coordinates (-6,-24):
LINE (-9, -28)-(-6, -24)

PRINT "Physical x-coordinate of the last point = " POINT(0)
PRINT "Physical y-coordinate of the last point = " POINT(1)
PRINT
PRINT "Window x-coordinate of the last point  = " POINT(2)
PRINT "Window y-coordinate of the last point  = " POINT(3)

END
```

Output

```
Physical x-coordinate of the last point = 511
Physical y-coordinate of the last point = 139

Window x-coordinate of the last point  = -6
Window y-coordinate of the last point  = -24
```

Given the **WINDOW** statement in the preceding example, the next four **PMAP** statements would print the output that follows:

```
' Map the window x-coordinate -6 to physical x and print:
PhysX% = PMAP(-6, 0)
PRINT PhysX%

' Map the window y-coordinate -24 to physical y and print:
PhysY% = PMAP(-24, 1)
PRINT PhysY%

' Map physical x back to view x and print:
WindowX% = PMAP(PhysX%, 2)
PRINT WindowX%

' Map physical y back to view y and print:
WindowY% = PMAP(PhysY%, 3)
PRINT WindowY%
```

Output

```
511
 139
-6
-24
```

Using Colors

If you have a Color Graphics Adapter (CGA), you can choose between the following two graphics modes only:

- Screen mode 2 has 640 x 200 high resolution with only one foreground and one background color. This is known as "monochrome," since all graphics output has the same color.
- Screen mode 1 has 320 x 200 medium resolution with 4 foreground colors and 16 background colors.

There is a tradeoff between color and clarity in the two screen modes supported by most color-graphics display adapter hardware. Depending on the graphics capability of your system, you may not have to sacrifice clarity to get a full range of color. However, this section focuses on screen modes 1 and 2.

Selecting a Color for Graphics Output

The following list shows where to put the *color* argument in the graphics statements discussed in previous sections of this chapter. This list also shows other options (such as **BF** with the **LINE** statement or *border* with the **VIEW** statement) that can have a different colors. (Please note that these do not give the complete syntax for some of these statements. This summary is intended to show how to use the *color* option in those statements that accept it.)

PSET (*x%,y%*),*color&*
PRESET (*x%,y%*),*color&*
LINE (*x1!,y1!*) (*x2!,y2!*),*color&*[,B[F]]
CIRCLE (*x!,y!*),*radius!*,*color&*
VIEW (*x1!,y1!*) (*x2!,y2!*),*color&*, *border&*

In screen mode 1, the *color* argument is a numeric expression with the value 0, 1, 2, or 3. Each of these values, known as an "attribute," represents a different color, as demonstrated by the following program:

```
' Draw an "invisible" line (same color as background):
LINE (10, 10)-(310, 10), 0

' Draw a light blue (cyan) line:
LINE (10, 30)-(310, 30), 1

' Draw a purple (magenta) line:
LINE (10, 50)-(310, 50), 2

' Draw a white line:
LINE (10, 70)-(310, 70), 3
END
```

As noted in the comments for the preceding example, a *color&* value of 0 produces no visible output, since it is always equal to the current background color. At first glance, this may not seem like such a useful color value, but, in fact, it is useful for erasing a figure without having to clear the entire screen or viewport, as shown in the next example:

```
SCREEN 1

CIRCLE (100, 100), 80, 2, , , 3    ' Draw an ellipse.
Pause$ = INPUT$(1)                 ' Wait for a key press.
CIRCLE (100, 100), 80, 0, , , 3    ' Erase the ellipse.
```

Changing the Foreground or Background Color

The preceding section describes how to use four different foreground colors for graphics output. You have a wider variety of colors in screen mode 1 for the screen's background: 16 in all.

In addition, you can change the foreground color by using a different "palette." In screen mode 1, there are two palettes, or groups of four colors. Each palette assigns a different color to the same attribute; so, for instance, in palette 1 (the default) the color associated with attribute 2 is magenta, while in palette 0 the color associated with attribute 2 is red. If you have a CGA, these colors are predetermined for each palette; that is, the color assigned to number 2 in palette 1 is always magenta, while the color assigned to number 2 in palette 0 is always red.

If you have an Enhanced Graphics Adapter (EGA) or Video Graphics Array (VGA), you can use the **PALETTE** statement to choose the color displayed by any attribute. For example, by changing arguments in a **PALETTE** statement, you could make the color displayed by attribute 1 green one time and brown the next. (See "Changing Colors with PALETTE and PALETTE USING" later in this chapter for more information on reassigning colors.)

In screen mode 1, the **COLOR** statement allows you to control the background color and the palette for the foreground colors. Here is the syntax for **COLOR** in screen mode 1:

COLOR [[*background&*]] [[,*palette%*]]

The *background&* argument is a numeric expression from 0 to 15, and *palette%* is a numeric expression equal to either 0 or 1. Table 5.1 shows the colors produced with the four different foreground numbers in each of the two palettes.

Table 5.1 Color Palettes in Screen Mode 1

Foreground color number	Color in palette 0	Color in palette 1
0	Current background color	Current background color
1	Green	Cyan (bluish green)
2	Red	Magenta (light purple)
3	Brown	White (light gray on some monitors)

Table 5.2 shows the colors produced with the 16 different background numbers.

Table 5.2 Background Colors in Screen Mode 1

Background color number	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta

Table 5.2 *Continued*

Background color number	Color
6	Brown (dark yellow on some monitors)
7	White (light gray on some monitors)
8	Dark gray (black on some monitors)
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Light yellow (may have greenish tinge on some monitors)
15	Bright white or very light gray

Example The following program shows all combinations of the two color palettes with the 16 different background screen colors. This program is in the file named COLORS.BAS on the Microsoft BASIC distribution disks.

```
SCREEN 1

Esc$ = CHR$(27)
' Draw three boxes and paint the interior
' of each box with a different color:
FOR ColorVal = 1 TO 3
   LINE (X, Y) -STEP(60, 50), ColorVal, BF
   X = X + 61
   Y = Y + 51
NEXT ColorVal

LOCATE 21, 1
PRINT "Press Esc to end."
PRINT "Press any other key to continue."

' Restrict additional printed output to the 23rd line:
VIEW PRINT 23 TO 23
DO
   PaletteVal = 1
   DO
```

```
    ' PaletteVal is either 1 or 0:
    PaletteVal = 1 - PaletteVal

    ' Set the background color and choose the palette:
    COLOR BackGroundVal, PaletteVal
    PRINT "Background ="; BackGroundVal;
    PRINT "Palette ="; PaletteVal;

    Pause$ = INPUT$(1)          ' Wait for a keystroke.
    PRINT
' Exit the loop if both palettes have been shown,
' or if the user pressed the Esc key:
LOOP UNTIL PaletteVal = 1 OR Pause$ = Esc$

    BackGroundVal = BackGroundVal + 1

' Exit this loop if all 16 background colors have
' been shown, or if the user pressed the Esc key:
LOOP UNTIL BackGroundVal > 15 OR Pause$ = Esc$

SCREEN 0                        ' Restore text mode and
WIDTH 80                        ' 80-column screen width.
```

Changing Colors with PALETTE and PALETTE USING

The preceding section shows how you can change the color displayed by an attribute simply by specifying a different palette in the **COLOR** statement. However, this restricts you to two fixed color palettes, with just four colors in each. Furthermore, each attribute can stand for only one of two possible colors; for example, attribute 1 can signify only green or cyan.

With an EGA or VGA, your choices are potentially much broader. (If you don't have an EGA or VGA, you may want to skip this section.) For instance, depending on the amount of video memory available to your computer, with a VGA you can choose from a palette with as many as 256,000 colors and assign those colors to 256 different attributes. Even an EGA allows you to display up to 16 different colors from a palette of 64 colors.

In contrast to the **COLOR** statement, the **PALETTE** and **PALETTE USING** statements give you a lot more flexibility in manipulating the available color palette. Using these statements, you can assign any color from the palette to any attribute. For example, after the following statement, the output of all graphics statements using attribute 4 appears in light magenta (color 13):

```
PALETTE 4, 13
```

This color change is instantaneous and affects not only subsequent graphics statements but any output already on the screen. In other words, you can draw and paint your screen, then switch the palette to achieve an immediate change of color, as shown by the following example:

```
SCREEN 8
LINE (50, 50)-(150, 150), 4   ' Draws a line in red.
SLEEP 1                       ' Pauses program.
PALETTE 4, 13                 ' Attribute 4 now means color
                              ' 13, so the line drawn in the
                              ' last statement is now light
                              ' magenta.
```

With the **PALETTE** statement's **USING** option, you can change the colors assigned to every attribute all at once.

Example In the following example, the **PALETTE USING** statement gives the illusion of movement on the screen by constantly rotating the colors displayed by attributes 1 through 15. This program is in the file named PALETTE.BAS on the Microsoft BASIC distribution disks.

```
DECLARE SUB InitPalette ()
DECLARE   SUB ChangePalette ()
DECLARE   SUB DrawEllipses ()

DEFINT A-Z
DIM SHARED PaletteArray(15)

SCREEN 8            ' 640 x 200 resolution; 16 colors

InitPalette         ' Initialize PaletteArray.
DrawEllipses        ' Draw and paint concentric ellipses.

DO                  ' Shift the palette until a key
   ChangePalette    ' is pressed.
LOOP WHILE INKEY$ = ""

END

' ====================== InitPalette ======================
'    This procedure initializes the integer array used to
'    change the palette.
' =========================================================

SUB InitPalette STATIC
   FOR I = 0 TO 15
      PaletteArray(I) =I
   NEXT   I
END SUB
```

```
' ====================== DrawEllipses ======================
'    This procedure draws 15 concentric ellipses and
'    paints the interior of each with a different color.
' ==========================================================

SUB DrawEllipses STATIC
   CONST ASPECT = 1 / 3
   FOR ColorVal = 15 TO1 STEP -1
      Radius = 20 * ColorVal
      CIRCLE (320, 100), Radius, ColorVal, , , ASPECT
      PAINT (320, 100),ColorVal
   NEXT
END SUB
```

```
' ====================== ChangePalette =====================
'  This procedure rotates the palette by one each time it
'  is called. For example, after the first call to
'  ChangePalette, PaletteArray(1) = 2, PaletteArray(2) = 3,
'  . . . , PaletteArray(14) = 15, and PaletteArray(15) = 1
' ==========================================================

SUB ChangePalette STATIC
   FOR I = 1 TO 15
      PaletteArray(I) =(PaletteArray(I) MOD 15) + 1
   NEXT   I
   PALETTE USING PaletteArray(0) ' Shift the color displayed
                                 ' by each of the attributes.
END SUB
```

Painting Shapes

The section "Drawing Boxes" earlier in this chapter shows how to draw a box with the **LINE** statement's **B** option, then paint the box by appending the F (for fill) option:

```
SCREEN 1

' Draw a square, then paint the interior with color 1
' (cyan in the default palette):
LINE (50, 50)-(110, 100), 1, BF
```

With BASIC's **PAINT** statement, you can fill any enclosed figure with any color you choose. **PAINT** also allows you to fill enclosed figures with your own custom patterns, such as stripes or checks, as shown in "Painting with Patterns: Tiling" later in this chapter.

Painting with Colors

To paint an enclosed shape with a solid color, use this form of the **PAINT** statement:

PAINT [[**STEP**]](*x!*,*y!*) [[, [[*paint*]], [[*bordercolor&*]]]]

Here, *x!*, *y!* are the coordinates of a point in the interior of the figure you want to paint, *paint* is the number for the color you want to paint with, and *bordercolor&* is the color number for the outline of the figure.

For example, the following program lines draw a circle in cyan, then paint the inside of the circle magenta:

```
SCREEN 1
CIRCLE (160, 100), 50, 1
PAINT (160, 100), 2, 1
```

The following three rules apply when painting figures:

- The coordinates given in the **PAINT** statement must refer to a point inside the figure. For example, any one of the following statements would have the same effect as the **PAINT** statements shown in the two preceding examples, since all of the coordinates identify points in the interior of the circle:

  ```
  PAINT (150, 90), 2, 1
  PAINT (170, 110), 2, 1
  PAINT (180, 80), 2, 1
  ```

  In contrast, since `(5, 5)` identifies a point outside the circle, the next statement would paint all of the screen except the inside of the circle, leaving it colored with the current background color:

  ```
  PAINT (5, 5), 2, 1
  ```

  If the coordinates in a **PAINT** statement specify a point right on the border of the figure, then no painting occurs:

  ```
  LINE (50, 50)-(150, 150), , B  ' Draw a box.
  PAINT (50, 100)   ' The point with coordinates
                    ' (50, 100) is on the top edge of the
                    ' box, so no painting occurs.
  ```

- The figure must be completely enclosed; otherwise, the paint color will "leak out," filling the entire screen or viewport (or any larger figure completely enclosing the first one).

  For example, in the following program, the **CIRCLE** statement draws an ellipse that is not quite complete (there is a small gap on the right side); the **LINE** statement then encloses the partial ellipse inside a box. Even though painting starts in the interior of the ellipse, the paint color flows through the gap and fills the entire box.

```
SCREEN 2
CONST PI = 3.141592653589#
CIRCLE (300, 100), 80, , 0, 1.9 * PI, 3
LINE (200, 10)-(400, 190), , B
PAINT (300, 100)
```

- If you are painting an object a different color from the one used to outline the object, you must use the *bordercolor&* option to tell **PAINT** where to stop painting.

  For example, the following program draws a triangle outlined in green (attribute 1 in palette 0) and then tries to paint the interior of the triangle red (attribute 2). However, since the **PAINT** statement doesn't indicate where to stop painting, it paints the entire screen red.

```
SCREEN 1
COLOR , 0

LINE (10, 25)-(310, 25), 1
LINE -(160, 175), 1
LINE -(10, 25), 1

PAINT (160, 100), 2
```

  Making the following change to the **PAINT** statement (choose red for the interior and stop when a border colored green is reached) produces the desired effect:

```
PAINT (160, 100), 2, 1
```

  Note that you don't have to specify a border color in the **PAINT** statement if the paint color is the same as the border color.

```
LINE (10, 25)-(310, 25), 1
LINE -(160, 175), 1
LINE -(10, 25), 1

PAINT (160, 100), 1
```

Painting with Patterns: Tiling

You can use the **PAINT** statement to fill any enclosed figure with a pattern; this process is known as "tiling." A "tile" is the pattern's basic building block. The process is identical to laying down tiles on a floor. When you use tiling, the argument *paint* in the syntax for **PAINT** is a string expression, rather than a number. While *paint* can be any string expression, a convenient way to define tile patterns uses the following form for *paint*:

CHR$**(*code1%*)+**CHR$(*code2%*)+**CHR$**(*code3%*)+...+**CHR$**(*coden%*)

Here, *code1%*, *code2%*, and so forth are 8-bit integers. See the following sections for an explanation of how these 8-bit integers are derived.

Pattern-Tile Size in Different Screen Modes

Each tile for a pattern is composed of a rectangular grid of pixels. This tile grid can have up to 64 rows in all screen modes. However, the number of pixels in each row depends on the screen mode.

The reason the length of each tile row varies according to the screen mode is because, although the number of bits in each row is fixed at 8 (the length of an integer), the number of pixels these 8 bits can represent decreases as the number of color attributes in a given screen mode increases. For example, in screen mode 2, which has only one color attribute, the number of bits per pixel is 1; in screen mode 1, which has four different attributes, the number of bits per pixel is 2; and in the EGA screen mode 7, which has 16 attributes, the number of bits per pixel is 4. The following formula allows you to compute the bits per pixel in any given screen mode:

$$bits\text{-}per\text{-}pixel = \log_2(numattributes)$$

Here, *numattributes* is the number of color attributes in that screen mode. (Online Help has this information.)

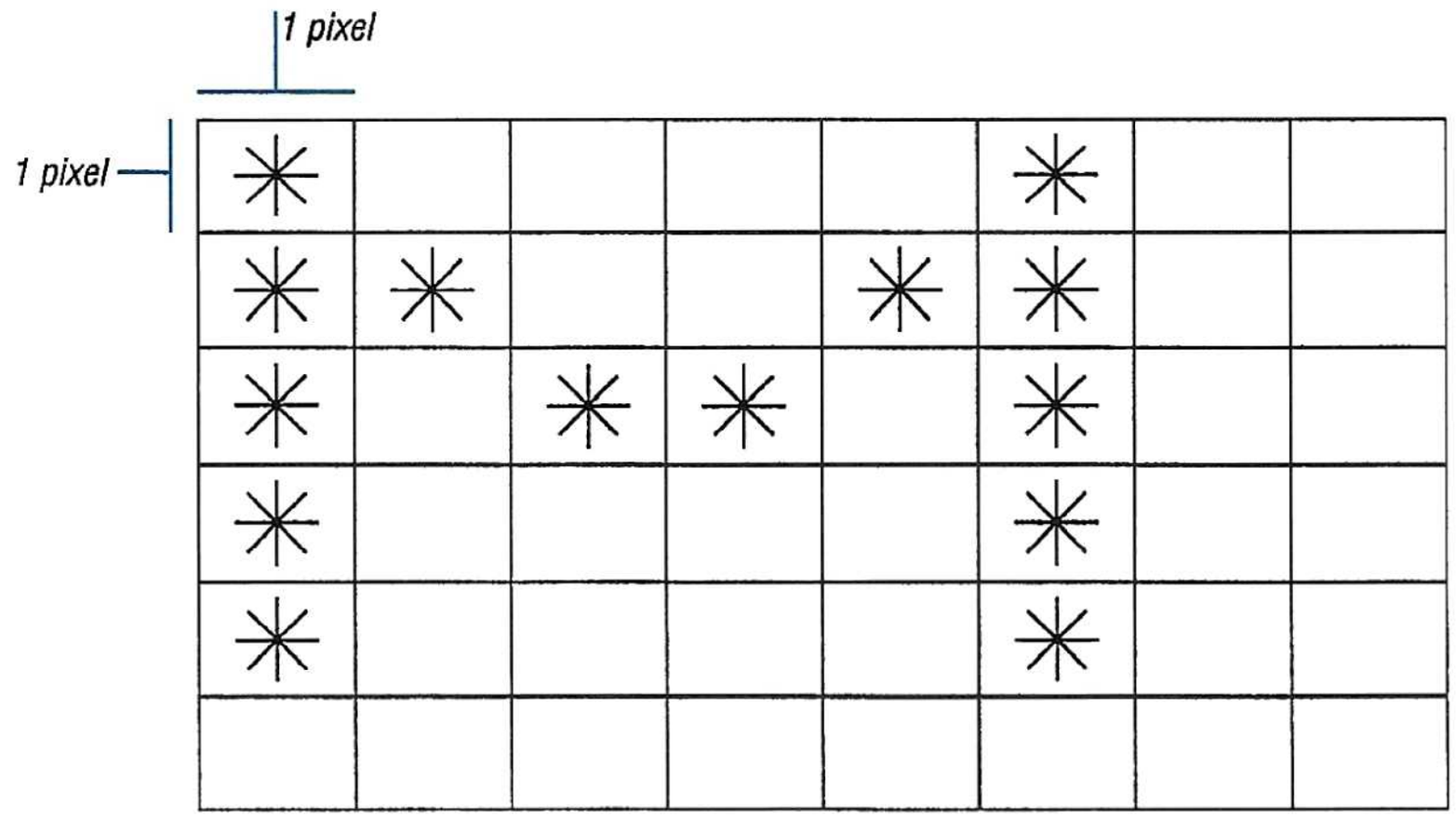
Thus, the length of a tile row is 8 pixels in screen mode 2 (8 bits divided by 1 bit per pixel), but only 4 pixels in screen mode 1 (8 bits divided by 2 bits per pixel).

The next three sections show the step-by-step process involved in creating a pattern tile. The following section shows how to make a monochrome pattern in screen mode 2. The section after that shows how to make a multicolored pattern in screen mode 1. Finally, if you have an EGA, read the third section to see how to make a multicolored pattern in screen mode 8.

Creating a Single-Color Pattern in Screen Mode 2

The following steps show how to define and use a pattern tile that resembles the letter "M":

1. Draw the pattern for a tile in a grid with eight columns and however many rows you need (up to 64). In this example, the tile has six rows; an asterisk (*) in a box means the pixel is on:



2. Next, translate each row of pixels to an 8-bit number, with a one meaning the pixel is on, and a zero meaning the pixel is off:

1	0	0	0	0	**1**	0	0
1	**1**	0	0	**1**	**1**	0	0
1	0	**1**	**1**	0	**1**	0	0
1	0	0	0	0	**1**	0	0
1	0	0	0	0	**1**	0	0
0	0	0	0	0	0	0	0

3. Convert the binary numbers given in step 2 to hexadecimal integers:

```
10000100 = &H84
11001100 = &HCC
10110100 = &HB4
10000100 = &H84
10000100 = &H84
00000000 = &H00
```

These integers do not have to be hexadecimal; they could be decimal or octal. However, binary to hexadecimal conversion is easier. To convert from binary to hexadecimal, read the binary number from right to left. Each group of four digits is then converted to its hexadecimal equivalent, as shown here:

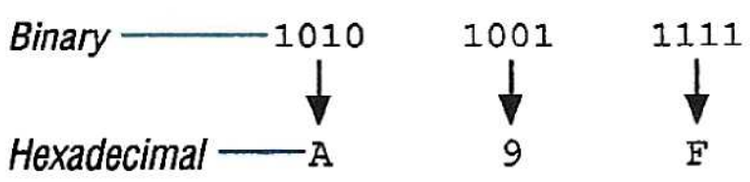


Table 5.3 lists 4-bit binary sequences and their hexadecimal equivalents.

4. Create a string by concatenating the characters with the ASCII values from step 3 (use the **CHR$** function to get these characters):

```
Tile$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4)
Tile$ = Tile$ + CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
```

5. Draw a figure and paint its interior using **PAINT** and the string argument from step 4:

```
PAINT (X, Y), Tile$
```

Table 5.3 Binary to Hexadecimal Conversion

Binary number	Hexadecimal number
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6

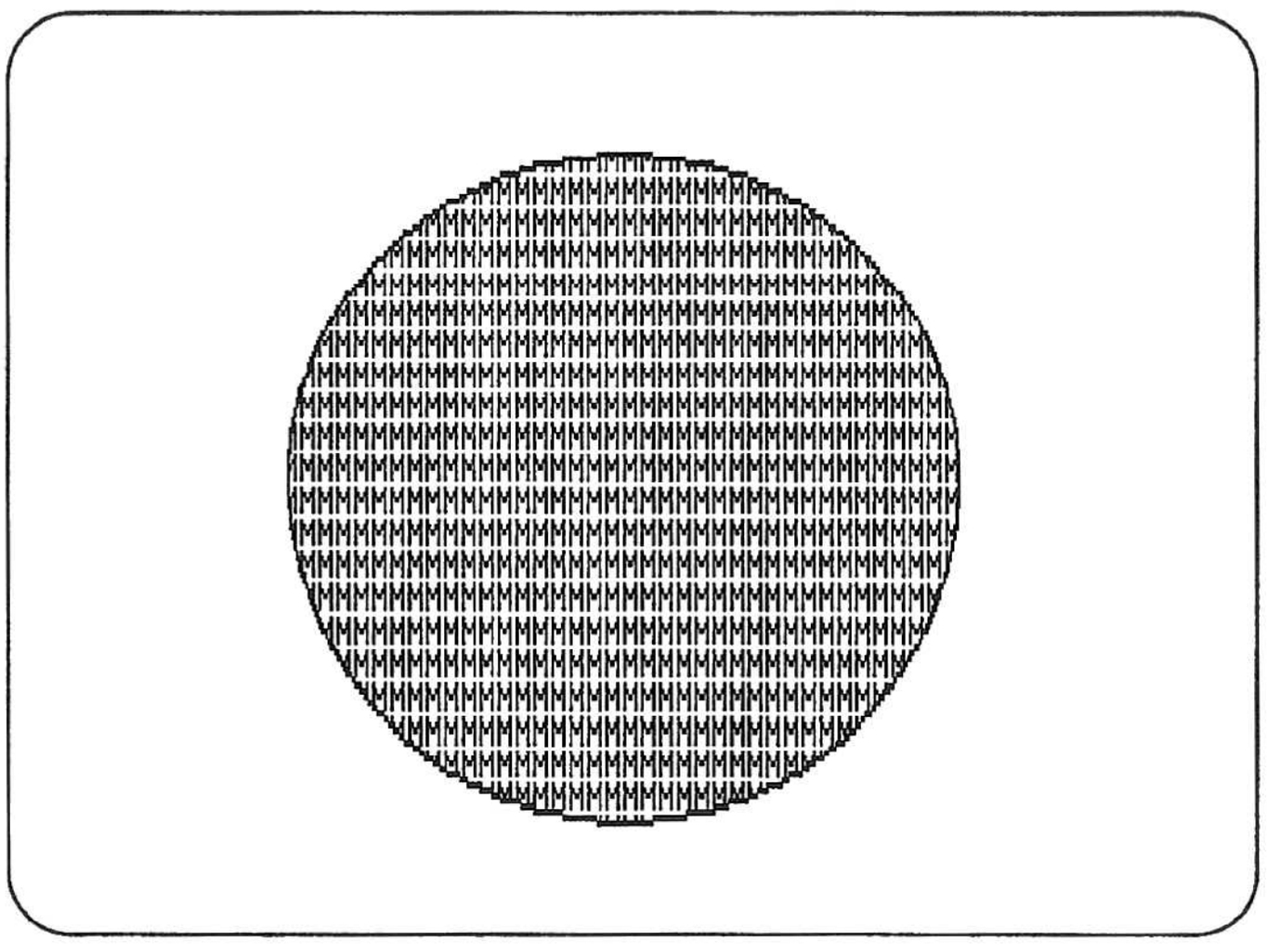
Table 5.3 *Continued*

Binary number	Hexadecimal number
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Example The following example draws a circle and then paints the circle's interior with the pattern created in the preceding steps:

```
SCREEN 2
CLS
Tile$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4)
Tile$ = Tile$ + CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
CIRCLE STEP(0, 0), 150
PAINT STEP(0, 0), Tile$
```

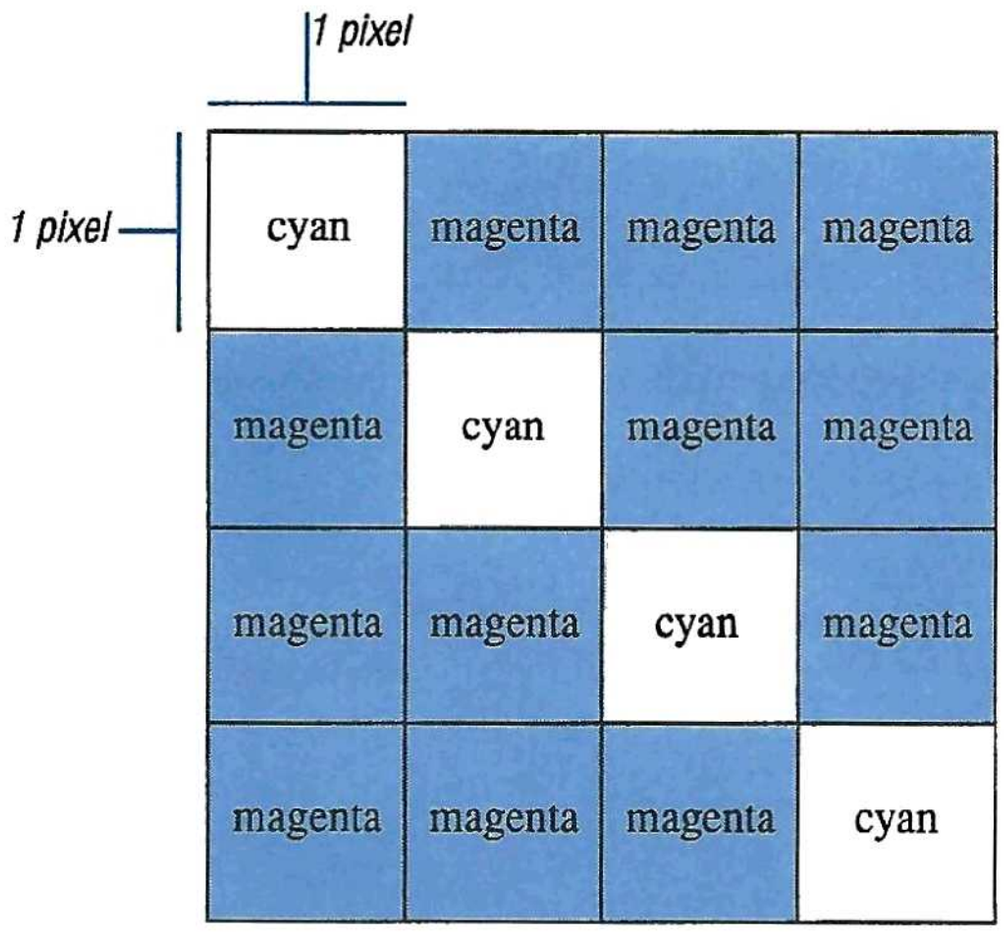
Output



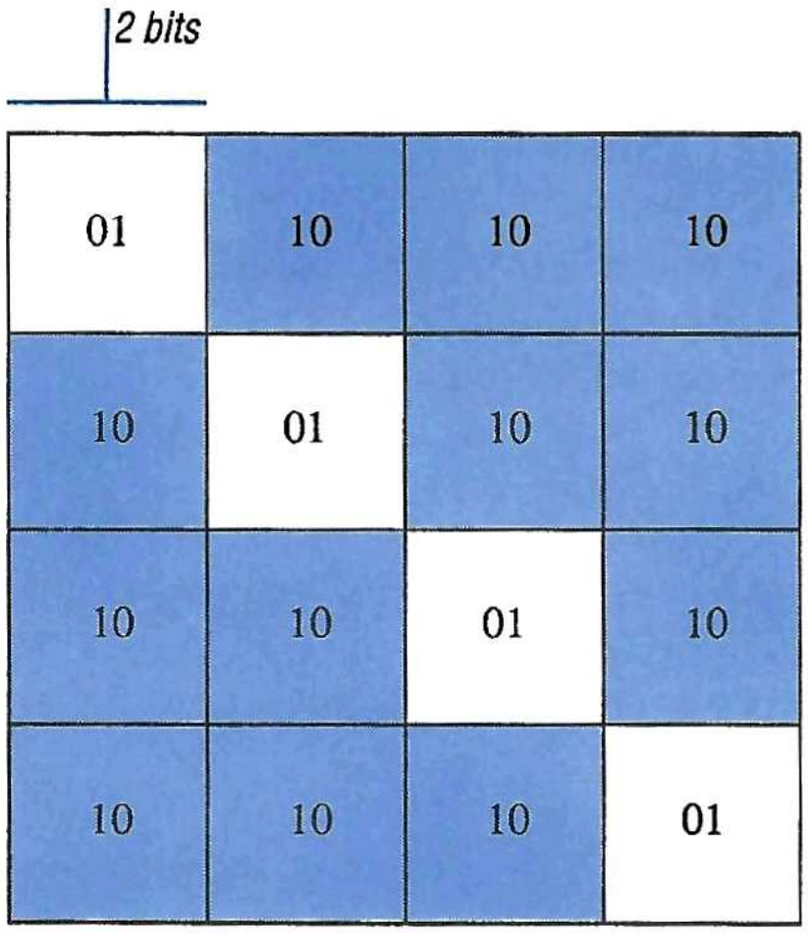
Creating a Multicolor Pattern in Screen Mode 1

The following steps show how to create a multicolor pattern consisting of alternating diagonal stripes of cyan and magenta (or green and red in palette 0):

1. Draw the pattern for a tile in a grid with four columns (four columns because each row of pixels is stored in an 8-bit integer and each pixel in screen mode 1 requires 2 bits) and however many rows you need (up to 64). In this example, the tile has four rows, as shown in the next diagram:



2. Convert the colors to their respective color numbers in binary notation, as shown by the following (be sure to use 2-bit values, so that 1 = binary 01 and 2 = binary 10):



3. Convert the binary numbers from step 2 to hexadecimal integers:

```
01101010 = &H6A
10011010 = &H9A
10100110 = &HA6
10101001 = &HA9
```

4. Create a string by concatenating the characters with the ASCII values from step 3 (use the **CHR$** function to get these characters):

```
Tile$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)
```

5. Draw a figure and paint its interior using **PAINT** and the string argument from step 4:

```
PAINT (X, Y), Tile$
```

The following program draws a triangle and then paints its interior with the pattern created in the preceding steps:

```
SCREEN 1

' Define a pattern:
Tile$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)
```

```
' Draw a triangle in white (color 3):
LINE (10, 25)-(310, 25)
LINE -(160, 175)
LINE -(10, 25)

' Paint the interior of the triangle with the pattern:
PAINT (160, 100), Tile$
```

Note that if the figure you want to paint is outlined in a color that is also contained in the pattern, you must give the *bordercolor&* argument with **PAINT** as shown by the following example; otherwise, the pattern spills over the edges of the figure:

```
SCREEN 1

' Define a pattern:
Tile$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + CHR$(&HA9)

' Draw a triangle in magenta (color 2):
LINE (10, 25)-(310, 25), 2
LINE -(160, 175), 2
LINE -(10, 25), 2

' Paint the interior of the triangle with the pattern,
' adding the border argument (, 2) to tell PAINT
' where to stop:
PAINT (160, 100), Tile$, 2
```

Sometimes, after painting a figure with a solid color or pattern, you may want to repaint that figure, or some part of it, with a new pattern. If the new pattern contains two or more adjacent rows that are the same as the figure's current background, you will find that tiling does not work. Instead, the pattern starts to spread, finds itself surrounded by pixels that are the same as two or more of its rows, then stops.

You can alleviate this problem by using the *background* argument with **PAINT** if there are at most two adjacent rows in your new pattern that are the same as the old background. **PAINT** with *background* has the following syntax:

PAINT [[**STEP**](*x!*,*y!*)[[,[*paint*]][,[*bordercolor&*]][,*background$*]]]

The *background$* argument is a string character of the form **CHR$**(*code%*) that specifies the rows in the pattern tile that are the same as the figure's current background. In essence, *background$* tells **PAINT** to skip over these rows when repainting the figure. The next example clarifies how this works:

```
SCREEN 1

' Define a pattern (two rows each of cyan, magenta, white):
Tile$ = CHR$(&H55) + CHR$(&H55) + CHR$(&HAA)
Tile$ = Tile$ + CHR$(&HAA) + CHR$(&HFF) + CHR$(&HFF)
```

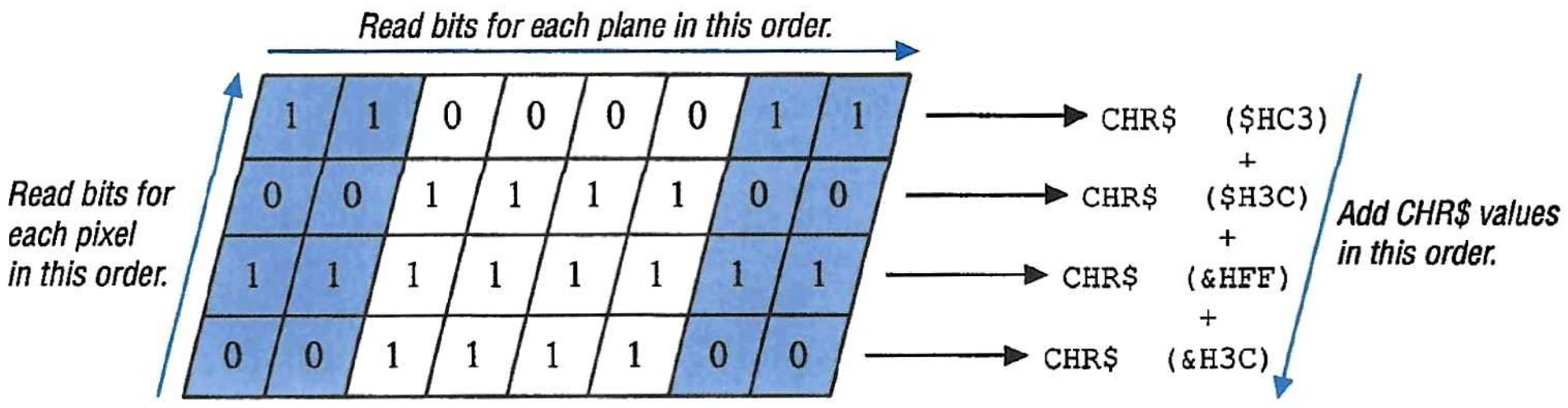
```
' Draw a triangle in white (color number 3):
LINE (10, 25)-(310, 25)
LINE -(160, 175)
LINE -(10, 25)

' Paint the interior magenta:
PAINT (160, 100), 2, 3

' Wait for a keystroke:
Pause$ = INPUT$(1)

' Since the background is already magenta, CHR$(&HAA) tells
' PAINT to skip over the magenta rows in the pattern tile:
PAINT (160, 100), Tile$, , CHR$(&HAA)
```

Creating a Multicolor Pattern in Screen Mode 8

In the EGA and VGA screen modes, it takes more than one 8-bit integer to define one row in a pattern tile. In these screen modes, a row is composed of several layers of 8-bit integers. This is because a pixel is represented three dimensionally in a stack of "bit planes" rather than sequentially in a single plane, as is the case with screen modes 1 and 2. For example, screen mode 8 has four of these bit planes. Each of the 4 bits per pixel in this screen mode is on a different plane.

The following steps diagram the process for creating a multicolor pattern consisting of rows of alternating yellow and magenta. Note how each row in the pattern tile is represented by 4 parallel bytes:

1. Define one row of pixels in the pattern tile. Each pixel in the row takes 4 bits, and each bit is in a different plane, as shown in the following:

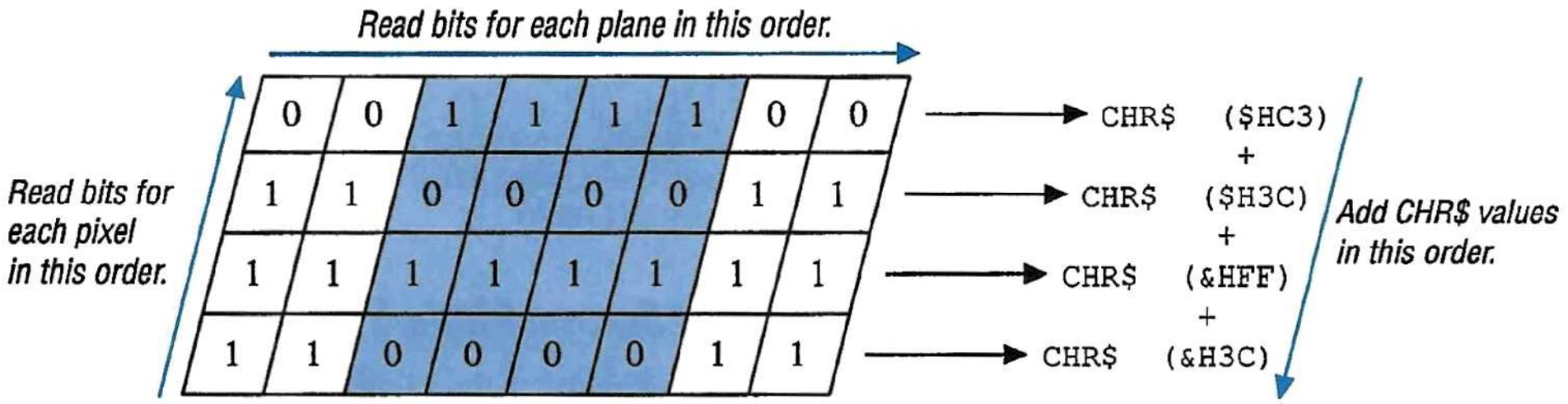


Key:	Color	Decimal	Binary
	Yellow	14	1110
	Magenta	5	0101

Add the **CHR$** values for all four bit planes to get one tile byte. This row is repeated in the pattern tile, so:

```
Row$(1) = Row$(2) = CHR$(&HC3) + CHR$(&H3C) + CHR$(&HFF) + CHR$(&H3C)
```

2. Define another row of pixels in the pattern tile, as follows:



Key:	Color	Decimal	Binary
	Yellow	14	1110
	Magenta	5	0101

This row is also repeated in the pattern tile, so:

```
Row$(3) = Row$(4) = CHR$(&H3C) + CHR$(&HC3) + CHR$(&HFF) + CHR$(&HC3)
```

Example The following example draws a box, then paints its interior with the pattern created in the preceding steps:

```
SCREEN 8
DIM Row$(1 TO 4)

' Two rows of alternating magenta and yellow:
Row$(1) = CHR$(&HC3) + CHR$(&H3C) + CHR$(&HFF) + CHR$(&H3C)
Row$(2) = Row$(1)

' Invert the pattern (two rows of alternating yellow
' and magenta):
Row$(3) = CHR$(&H3C) + CHR$(&HC3) + CHR$(&HFF) + CHR$(&HC3)
Row$(4) = Row$(3)
' Create a pattern tile from the rows defined above:
FOR I% = 1 TO 4
   Tile$ = Tile$ + Row$(I%)
NEXT I%

' Draw box and fill it with the pattern:
LINE (50, 50)-(570, 150), , B
PAINT (320, 100), Tile$
```

DRAW: A Graphics Macro Language

The **DRAW** statement is a miniature language by itself. It draws and paints images on the screen using a set of one- or two-letter commands, known as "macros," embedded in a string expression.

DRAW offers the following advantages over the other graphics statements discussed so far:

- The macro string argument to **DRAW** is compact: a single, short string can produce the same output as several **LINE** statements.
- Images created with **DRAW** can easily be scaled—that is, enlarged or reduced in size—by using the **S** macro in the macro string.
- Images created with **DRAW** can be rotated any number of degrees by using the **TA** macro in the macro string.

Consult online Help for more information.

Example The following program gives a brief introduction to the movement macros **U**, **D**, **L**, **R**, **E**, **F**, **G**, and **H**; the "plot/don't plot" macro **B**; and the color macro **C**. This program draws horizontal, vertical, and diagonal lines in different colors, depending on which direction key on the numeric keypad (Up Arrow, Down Arrow, Left Arrow, PgUp, PgDn, and so on) is pressed.

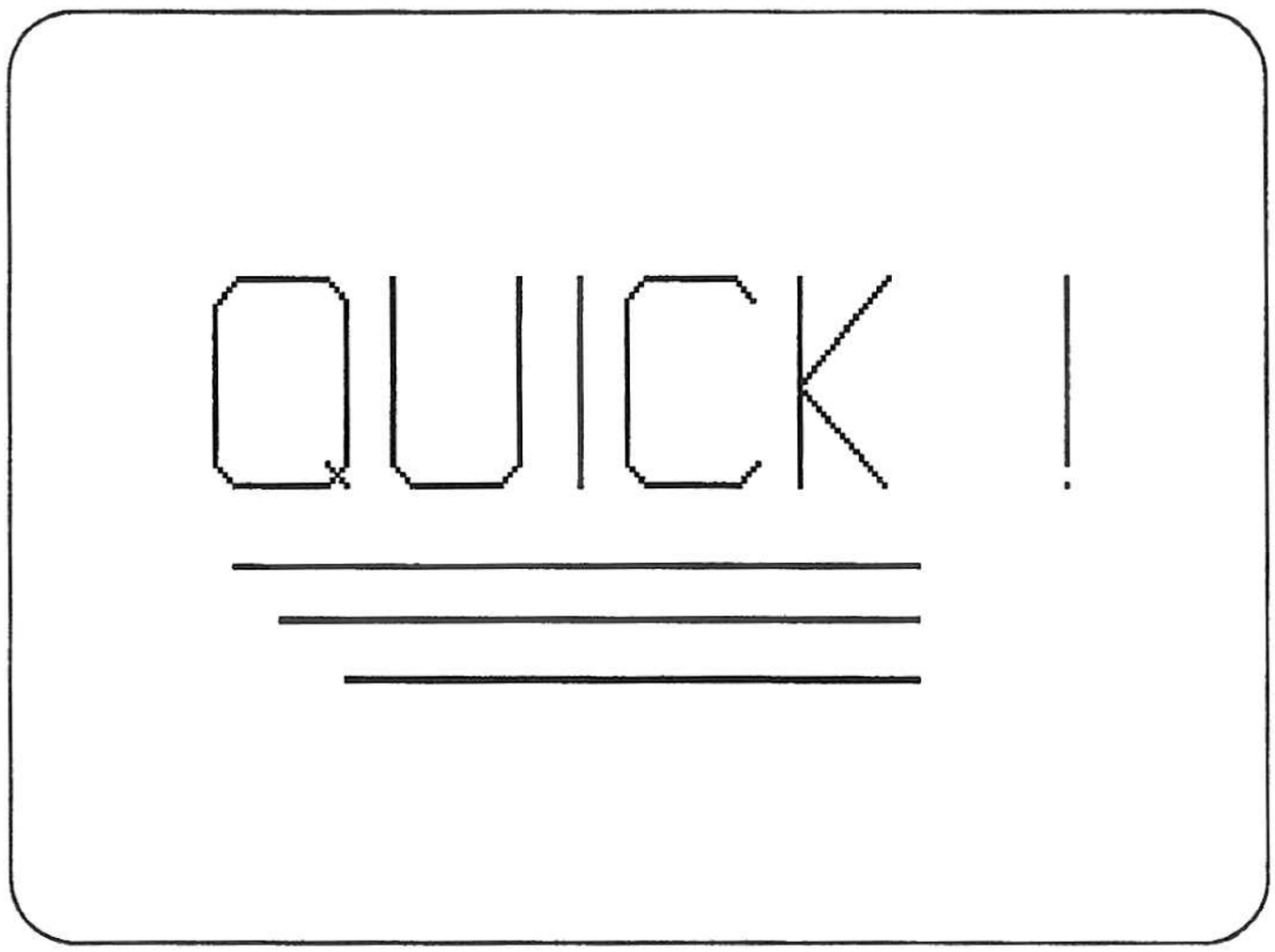
This program is in the file named PLOTTER.BAS on the Microsoft BASIC distribution disks.

```
' Values for keys on the numeric keypad and the Spacebar:
CONST UP = 72, DOWN = 80, LFT = 75, RGHT = 77
CONST UPLFT = 71, UPRGHT = 73, DOWNLFT = 79, DOWNRGHT = 81
CONST SPACEBAR = " "

' Null$ is the first character of the two-character INKEY$
' value returned for direction keys such as Up and Down:
Null$ = CHR$(0)
' Plot$ = "" means draw lines; Plot$ = "B" means
' move graphics cursor, but don't draw lines:
Plot$ = ""

PRINT "Use the cursor movement keys to draw lines."
PRINT "Press Spacebar to toggle line drawing on and off."
PRINT "Press <ENTER> to begin. Press q to end the program."
DO : LOOP WHILE INKEY$ = ""

SCREEN 1

DO
   SELECT CASE KeyVal$
      CASE Null$ + CHR$(UP)
         DRAW Plot$ + "C1 U2"
      CASE Null$ + CHR$(DOWN)
         DRAW Plot$ + "C1 D2"
      CASE Null$ + CHR$(LFT)
         DRAW Plot$ + "C2 L2"
      CASE Null$ + CHR$(RGHT)
         DRAW Plot$ + "C2 R2"
      CASE Null$ + CHR$(UPLFT)
         DRAW Plot$ + "C3 H2"
      CASE Null$ + CHR$(UPRGHT)
         DRAW Plot$ + "C3 E2"
      CASE Null$ + CHR$(DOWNLFT)
         DRAW Plot$ + "C3 G2"
      CASE Null$ + CHR$(DOWNRGHT)
         DRAW Plot$ + "C3 F2"
      CASE SPACEBAR
         IF Plot$ = "" THEN Plot$ = "B " ELSE Plot$ = ""
      CASE ELSE
    ' The user pressed some key other than one of the
    ' direction keys, the Spacebar, or "q," so
    ' don't do anything.
   END SELECT
   KeyVal$ = INKEY$
```

```
LOOP UNTIL KeyVal$ = "q"

SCREEN 0, 0             ' Restore the screen to 80-column
WIDTH 80                ' text mode and end.
END
```

Output

Here's a sample sketch created with this program.



Basic Animation Techniques

Using only the graphics statements covered in earlier sections, you can do simple animation of objects on the screen. For instance, you can first draw a circle with **CIRCLE**, then redraw it with the background color to erase it, and finally recalculate the circle's center point and draw it in a new location.

This technique works well enough for simple figures, but its disadvantages become apparent when animating more complex images. Even though the graphics statements are very fast, you can still notice the lag. Moreover, it is not possible to preserve the background with this method: when you erase the object, you also erase whatever was already on the screen.

Two statements allow you to do high-speed animation: **GET** and **PUT**. You can create an image using output from statements such as **PSET**, **LINE**, **CIRCLE**, or **PAINT**, then take a "snapshot" of that image with **GET**, copying the image to memory. With **PUT**, you can then reproduce the image stored with **GET** anywhere on the screen or viewport.

Saving Images with GET

After you have created the original image on the screen, you need to calculate the x- and y-coordinates of a rectangle large enough to hold the entire image. You then use **GET** to copy the entire rectangle to memory. The syntax for the graphics **GET** statement is as follows:

GET [[**STEP**]](*x1!*,*y1!*)–[[**STEP**]](*x2!*,*y2!*),*arrayname#*

The arguments (*x1!*, *y1!*) and (*x2!*, *y2!*) give the coordinates of a rectangle's upper-left and lower-right corners. The argument *arrayname#* refers to any numeric array. The size specified in a **DIM** statement for *arrayname#* depends on the following three factors:

- The height and width of the rectangle enclosing the image
- The screen mode chosen for graphics output
- The type of the array (integer, long integer, single precision, or double precision)

Note

Although the array used to store images can have any numeric type, it is strongly recommended that you use only integer arrays. All possible graphics patterns on the screen can be represented by integers. This is not the case, however, with single-precision or double-precision real numbers. Some graphics patterns are not valid real numbers, and it could lead to unforeseen results if these patterns were stored in a real-number array.

The formula for calculating the size in bytes of *arrayname#* is as follows:

size-in-bytes = 4 + *height* * *planes* * **INT**((*width* * *bits-per-pixel*/*planes* + 7)/8)

In the preceding syntax, *height* and *width* are the dimensions, in number of pixels, of the rectangle to get, and the value for *bits-per-pixel* depends on the number of colors available in the given screen mode. More colors mean more bits are required to define each pixel. In screen mode 1, two bits define a pixel, while in screen mode 2, one bit is enough. (See "Pattern-Tile Size in Different Screen Modes" earlier in this chapter for the general formula for *bits-per-pixel*.) The following list shows the value for *planes* for each of the screen modes:

Screen mode	Number of bit planes
1, 2, 11, and 13	1
9 (64K of video memory) and 10	2
7, 8, 9 (more than 64K of video memory), and 12	4

To get the number of elements that should be in the array, divide the *size-in-bytes* by the number of bytes for one element in the array. This is where the type of the array comes into play. If it is an integer array, each element takes 2 bytes of memory (the size of an integer), so *size-in-bytes* should be divided by two to get the actual size of the array. Similarly, if it is a long integer array, *size-in-bytes* should be divided by four (since one long integer requires 4 bytes of memory), and so on. If it is single precision, divide by four; if it is double precision, divide by eight.

The following steps show how to calculate the size of an integer array large enough to hold a rectangle in screen mode 1 with coordinates (10, 40) for the upper-left corner and (90, 80) for the lower-right corner:

1. Calculate the height and width of the rectangle:

   RectHeight = **ABS**(*y2* – *y1*) + 1 = 80 – 40 + 1 = 41
   RectWidth = **ABS**(*x2* – *x1*) + 1 = 90 – 10 + 1 = 81

   Remember to add one after subtracting *y1* from *y2* or *x1* from *x2*. For example, if *x1* = 10 and *x2* = 20, then the width of the rectangle is 20 – 10 + 1, or 11.

2. Calculate the size in bytes of the integer array:

   ByteSize = 4 + RectHeight * **INT**((RectWidth * BitsPerPixel + 7) / 8)
   = 4 + 41 * **INT**((81 * 2 + 7) / 8)
   = 4 + 41 * **INT**(169 / 8)
   = 4 + 41 * 21
   = 865

3. Divide the size in bytes by the bytes per element (two for integers) and round the result up to the nearest whole number:

   865 / 2 = 433

Therefore, if the name of the array is `Image()`, the following **DIM** statement ensures that `Image` is big enough to copy the pixel information in the rectangle:

```
DIM Image (1 TO 433) AS INTEGER
```

Note Although the **GET** statement uses view coordinates after a **WINDOW** statement, you must use physical coordinates to calculate the size of the array used in **GET**. (See the section "Redefining Viewport Coordinates with WINDOW" earlier in this chapter for more information on **WINDOW** and how to convert view coordinates to physical coordinates.)

Note that the steps outlined previously give the minimum size required for the array; however, any larger size will do. For example, the following statement also works:

```
DIM Image (1 TO 500) AS INTEGER
```

The following program draws an ellipse and paints its interior. A **GET** statement copies the rectangular area containing the ellipse into memory. (The following section, "Moving Images with PUT" shows how to use the **PUT** statement to reproduce the ellipse in a different location.)

```
SCREEN 1

' Dimension an integer array large enough
' to hold the rectangle:
DIM Image (1 TO 433) AS INTEGER

' Draw an ellipse inside the rectangle, using magenta for
' the outline and painting the interior white:
CIRCLE (50, 60), 40, 2, , , .5
PAINT (50, 60), 3, 2

' Store the image of the rectangle in the array:
GET (10, 40)-(90, 80), Image
```

Moving Images with PUT

While the **GET** statement allows you to take a snapshot of an image, **PUT** allows you to paste that image anywhere you want on the screen. A statement of the following form copies the rectangular image stored in *arrayname#* back to the screen and places its upper-left corner at the point with coordinates (*x!*, *y!*):

PUT(*x!*,*y!*),*arrayname#* [[,*actionverb*]]

Note that only one coordinate pair appears in **PUT**.

If a **WINDOW** statement appears in the program before **PUT**, the coordinates *x* and *y* refer to the lower-left corner of the rectangle. **WINDOW SCREEN**, however, does not have this effect; that is, after **WINDOW SCREEN**, *x* and *y* still refer to the upper-left corner of the rectangle.

For example, adding the next line to the last example in the section "Saving Images with **GET**" causes an exact duplicate of the painted ellipse to appear on the right side of the screen much more quickly than redrawing and repainting the same figure with **CIRCLE** and **PAINT**:

```
PUT (200, 40), Image
```

Take care not to specify coordinates that would put any part of the image outside the screen or active viewport, as in the following statements:

```
SCREEN 2
.
.
.
' Rectangle measures 141 pixels x 91 pixels:
GET (10, 10)-(150, 100), Image
PUT (510, 120), Image
```

Unlike other graphics statements such as **LINE** or **CIRCLE**, **PUT** does not clip images lying outside the viewport. Instead, it produces an error message `Illegal function call.`

One of the other advantages of the **PUT** statement is that you can control how the stored image interacts with what is already on the screen by using the argument *actionverb*. The *actionverb* argument can be one of the following options: **PSET**, **PRESET**, **AND**, **OR**, or **XOR**.

If you do not care what happens to the existing screen background, use the **PSET** option, since it transfers an exact duplicate of the stored image to the screen and overwrites anything that was already there.

Table 5.4 shows how other options affect the way the **PUT** statement causes pixels in a stored image to interact with pixels on the screen. In this table, 1 means a pixel is on and 0 means a pixel is off.

Table 5.4 The Effect of Different Action Options in Screen Mode 2

Action option	Pixel in stored image	Pixel on screen before PUT statement	Pixel on screen after PUT statement
PSET	0	0	0
	0	1	0
	1	0	1
	1	1	1
PRESET	0	0	1
	0	1	1
	1	0	0
	1	1	0
AND	0	0	0
	0	1	0
	1	0	0
	1	1	1
OR	0	0	0
	0	1	1
	1	0	1
	1	1	1

Table 5.4 *Continued*

Action option	Pixel in stored image	Pixel on screen before PUT statement	Pixel on screen after PUT statement
XOR	0	0	0
	0	1	1
	1	0	1
	1	1	0

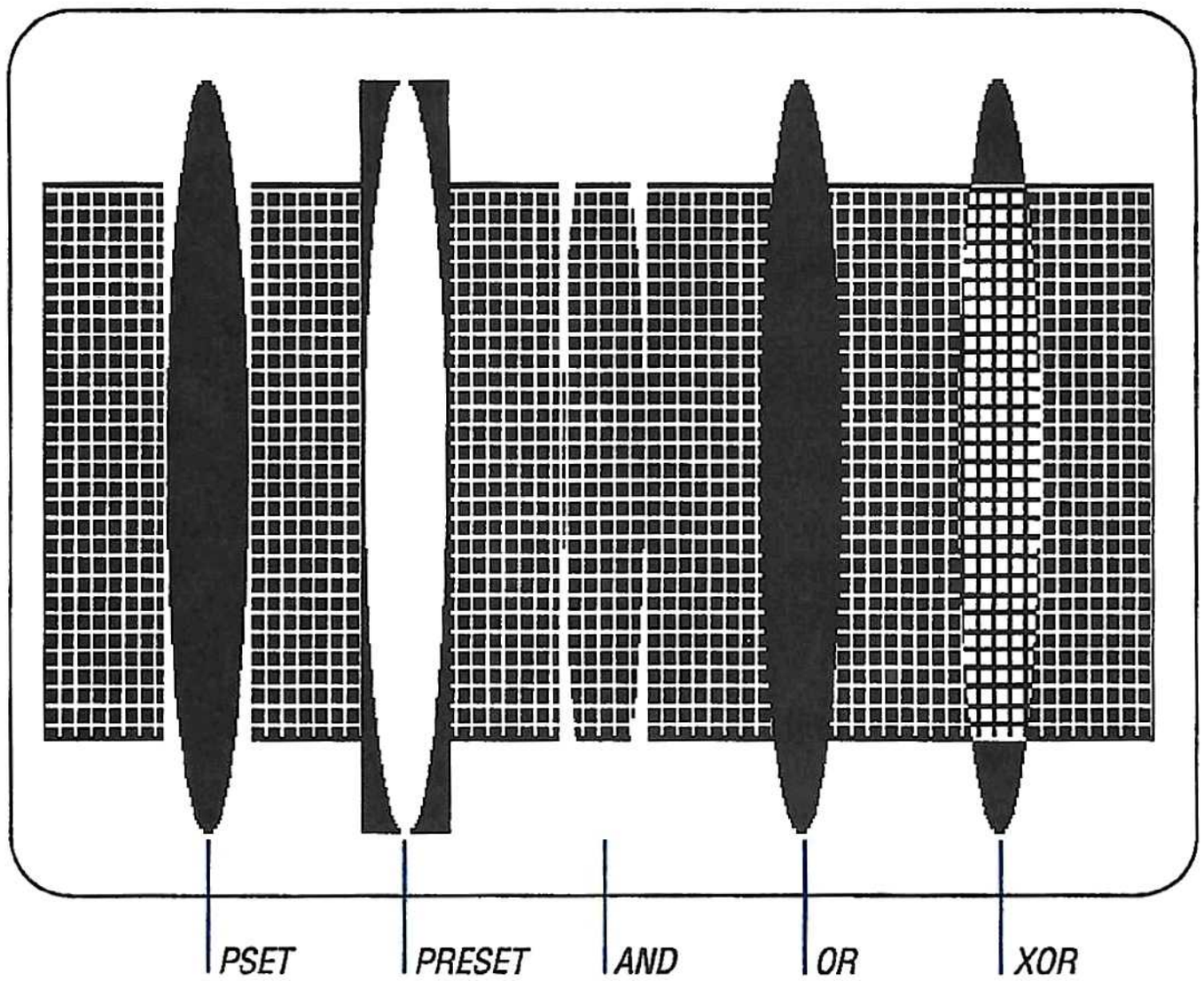
As you can see, these options cause a **PUT** statement to treat pixels the same way logical operators treat numbers. The **PRESET** option is like the logical operator **NOT** in that it inverts the pixels in the stored image, regardless of what was on the screen. The options **AND**, **OR**, and **XOR** are identical to the logical operators with the same names; for example, the following logical operation gives 0 as its result, just as using the **XOR** option turns a pixel off when the pixel in the image and the pixel in the background are on:

```
1 XOR 1
```

Example The output from the following program shows the same image superimposed over a filled rectangle using each of the five options discussed previously:

```
SCREEN 2

DIM CircImage (1 TO 485) AS INTEGER

' Draw and paint an ellipse then store its image with GET:
CIRCLE (22, 100), 80, , , , 4
PAINT (22, 100)
GET (0, 20)-(44, 180), CircImage
CLS

' Draw a box and fill it with a pattern:
LINE (40, 40)-(600, 160), , B
Pattern$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (50, 50), Pattern$
' Put the images of the ellipse over the box
' using the different action options:
PUT (100, 20), CircImage, PSET
PUT (200, 20), CircImage, PRESET
PUT (300, 20), CircImage, AND
PUT (400, 20), CircImage, OR
PUT (500, 20), CircImage, XOR
```

Output



In screen modes supporting color, the options **PRESET**, **AND**, **OR**, and **XOR** produce a more complicated interaction, since color involves more than simply turning a pixel on or off. However, the analogy made between these options and logical operators still holds in these modes. For example, if the current pixel on the screen is color 1 (cyan in palette 1) and the pixel in the overlaid image is color 2 (magenta in palette 1), then the color of the resulting pixel after a **PUT** statement depends on the option, as shown for just 6 of the 16 different combinations of image color and background color in Table 5.5.

Table 5.5 The Effect of Different Action Options on Color in Screen Mode 1 (Palette 1)

Action option	Pixel color in stored image	Pixel color on screen before PUT statement	Pixel color on screen after PUT statement
PSET	10 (magenta)	01 (cyan)	10 (magenta)
PRESET	10 (magenta)	01 (cyan)	01 (cyan)
AND	10 (magenta)	01 (cyan)	00 (black)
OR	10 (magenta)	01 (cyan)	11 (white)
XOR	10 (magenta)	01 (cyan)	11 (white)

In palette 1, cyan is assigned to attribute 1 (01 binary), magenta is assigned to attribute 2 (10 binary), and white is assigned to attribute 3 (11 binary). If you have an EGA, you can use the **PALETTE** statement to assign different colors to the attributes 1, 2, and 3.

Animation with GET and PUT

One of the most useful things that can be done with the **GET** and **PUT** statements is animation. The two options best suited for animation are **XOR** and **PSET**. Animation done with **PSET** is faster; but as shown by the output from the last program, **PSET** erases the screen background. In contrast, **XOR** is slower but restores the screen background after the image is moved. Animation with **XOR** is done with the following four steps:

1. Put the object on the screen with **XOR**.
2. Calculate the new position of the object.
3. Put the object on the screen a second time at the old location, using **XOR** again, this time to remove the old image.
4. Go to step 1, but this time put the object at the new location.

Movement done with these four steps leaves the background unchanged after step 3. Flicker can be reduced by minimizing the time between steps 4 and 1 and by making sure that there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, use the **PSET** option for animation. The idea is to leave a border around the image when you copy it with the **GET** statement. If this border is as large as or larger than the maximum distance the object will move, then each time the image is put in a new location, the border erases all traces of the image in the old location. This method can be faster than the method using **XOR** described previously, since only one **PUT** statement is required to move an object (although you must move a larger image).

The following example shows how to use **PUT** with the **PSET** option to produce the effect of a ball bouncing off the bottom and sides of a box. Note in the output that follows how the rectangle containing the ball, specified in the **GET** statement, erases the filled box and the printed message.

Examples

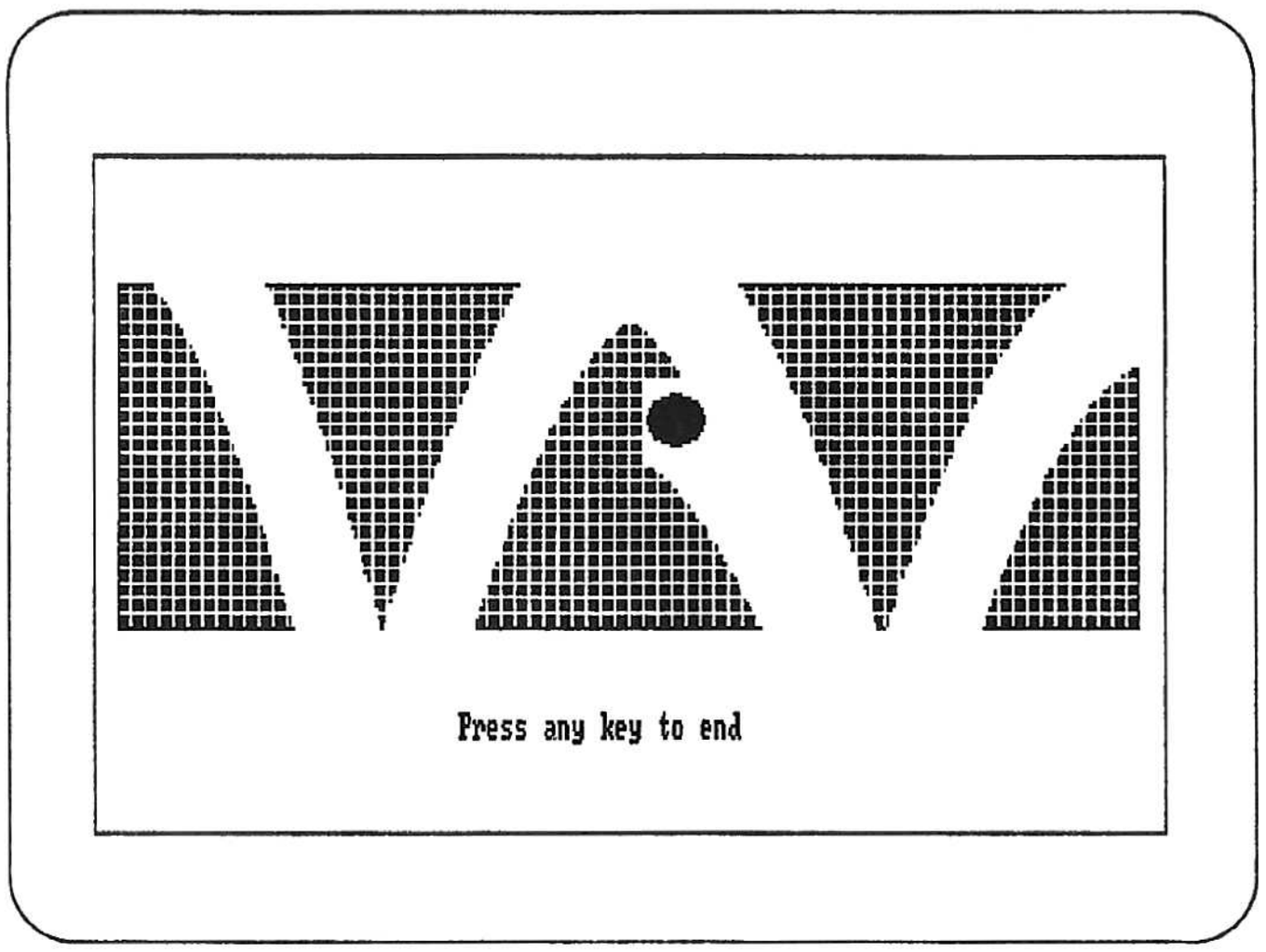
This program is in the file named BALLPSET.BAS on the Microsoft BASIC distribution disks.

```
DECLARE FUNCTION GetArraySize (WLeft, WRight, WTop, WBottom)

SCREEN 2

' Define a viewport and draw a border around it:
VIEW (20, 10)-(620, 190),,1

CONST PI = 3.141592653589#
```

```
' Redefine the coordinates of the viewport with window
' coordinates:
WINDOW (-3.15, -.14)-(3.56, 1.01)

' Arrays in program are now dynamic:
' $DYNAMIC

' Calculate the window coordinates for the top and bottom of a
' rectangle large enough to hold the image that will be
' drawn with CIRCLE and PAINT:
WLeft = -.21
WRight = .21
WTop = .07
WBottom = -.07

' Call the GetArraySize function,
' passing it the rectangle's window coordinates:
ArraySize% = GetArraySize(WLeft, WRight, WTop, WBottom)

DIM Array (1 TO ArraySize%) AS INTEGER

' Draw and paint the circle:
CIRCLE (0, 0), .18
PAINT (0, 0)

' Store the rectangle in Array:
GET (WLeft, WTop)-(WRight, WBottom), Array
CLS
' Draw a box and fill it with a pattern:
LINE (-3, .8)-(3.4, .2), , B
Pattern$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (0, .5), Pattern$

LOCATE 21, 29
PRINT "Press any key to end."

' Initialize loop variables:
StepSize = .02
StartLoop = -PI
Decay = 1

DO
   EndLoop = -StartLoop
   FOR X = StartLoop TO EndLoop STEP StepSize

      ' Each time the ball "bounces" (hits the bottom of the
      ' viewport), the Decay variable gets smaller, making
      ' the height of the next bounce smaller:
      Y = ABS(COS(X)) * Decay - .14
      IF Y < -.13 THEN Decay = Decay * .9
```

```
    ' Stop if key pressed or Decay less than .01:
    Esc$ = INKEY$
    IF Esc$ <> "" OR Decay < .01 THEN EXIT FOR

    ' Put the image on the screen. The StepSize offset is
    ' smaller than the border around the circle. Thus,
    ' each time the image moves, it erases any traces
    ' left from the previous PUT (and also erases anything
    ' else on the screen):
    PUT (X, Y), Array, PSET
NEXT X

' Reverse direction:
StepSize = -StepSize
   StartLoop = -StartLoop
LOOP UNTIL Esc$ <> "" OR Decay < .01

END

FUNCTION GetArraySize (WLeft, WRight, WTop, WBottom) STATIC

    ' Map the window coordinates passed to this function to
    ' their physical coordinate equivalents:
    VLeft = PMAP(WLeft, 0)
    VRight = PMAP(WRight, 0)
    VTop = PMAP(WTop, 1)
    VBottom = PMAP(WBottom, 1)
    ' Calculate the height and width in pixels
    ' of the enclosing rectangle:
    RectHeight = ABS(VBottom - VTop) + 1
    RectWidth = ABS(VRight - VLeft) + 1

    ' Calculate size in bytes of array:
    ByteSize = 4 + RectHeight * INT((RectWidth + 7) / 8)

    ' Array is integer, so divide bytes by two:
    GetArraySize = ByteSize \ 2 + 1
END FUNCTION
```

Output



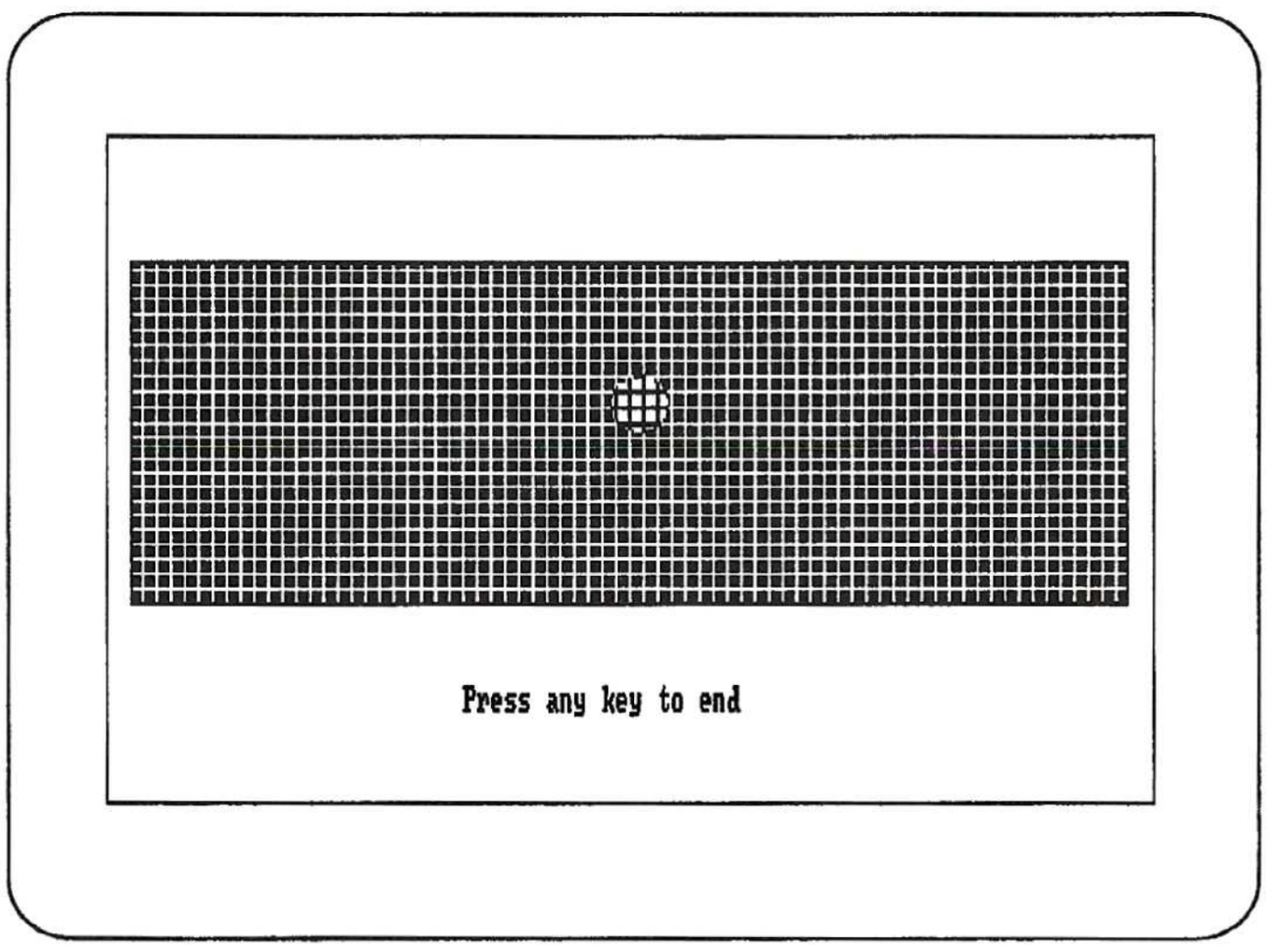
Contrast the preceding program with the next program, which uses **PUT** with **XOR** to preserve the screen's background, according to the steps outlined earlier. Note how the rectangle containing the ball is smaller than in the preceding program, since it is not necessary to leave a border. Also note that two **PUT** statements are required, one to make the image visible and another to make it disappear. Finally, observe the empty **FOR...NEXT** delay loop between the **PUT** statements; this loop reduces the flicker that results from the image appearing and disappearing too rapidly.

This program is in the file named BALLXOR.BAS on the Microsoft BASIC distribution disks.

```
.
.
' The rectangle is smaller than the one in the previous
' program, which means Array is also smaller:
WLeft = -.18
WRight = .18
WTop = .05
WBottom = -.05
.
.
.
DO
   EndLoop = -StartLoop
   FOR X = StartLoop TO EndLoop STEP StepSize
      Y = ABS(COS(X)) * Decay - .14
```

```
' The first PUT statement places the image
' on the screen:
PUT (X,Y), Array, XOR

' Use an empty FOR...NEXT loop to delay
' the program and reduce image flicker:
FOR I = 1 TO 5: NEXT I

IF Y < -.13 THEN Decay = Decay * .9
Esc$ = INKEY$
IF Esc$ <> "" OR Decay < .01 THEN EXIT FOR

' The second PUT statement erases the image and
' restores the background:
   PUT (X, Y), Array, XOR
NEXT X

StepSize = -StepSize
StartLoop = -StartLoop
LOOP UNTIL Esc$ <> "" OR Decay < .01

END
.
.
.
```

Output



Animating with Screen Pages

This section describes an animation technique that utilizes multiple pages of your computer's video memory.

Pages in video memory are analogous to pages in a book. Depending on the graphics capability of your computer, what you see displayed on the screen may only be part of the video memory available — just as what you see when you open a book is only part of the book. However, unlike a book, the unseen pages of your computer's video memory can be active; that is, while you are looking at one page on the screen, graphics output can be taking place on the others. It's as if the author of a book were still writing new pages even as you were reading the book.

The area of video memory visible on the screen is called the "visual page," while the area of video memory where graphics statements put their output is called the "active page." The **SCREEN** statement allows you to select visual and active screen pages with the following syntax:

SCREEN[[*mode%*]] [[,[[*activepage%*]][[,*visiblepage%*]]]]

In this syntax, *activepage%* is the number of the active page, and *visiblepage%* is the number of the visual page. The active page and the visual page can be one and the same (and are by default when the *activepage%* or *visiblepage%* arguments are not used with **SCREEN**, in which case the value of both arguments is 0).

You can animate objects on the screen by selecting a screen mode with more than one video memory page, then alternating the pages, sending output to one or more active pages while displaying finished output on the visual page. This technique makes an active page visible only when output to that page is complete. Since the viewer sees only a finished image, the display is instantaneous.

Example

The following program demonstrates the technique discussed previously. It selects screen mode 7, which has two pages, then draws a cube with the **DRAW** statement. This cube is then rotated through successive 15° angles by changing the value of the `TA` macro in the string used by **DRAW**. By swapping the active and visual pages back and forth, this program always shows a completed cube while a new one is being drawn.

This program is in the file named CUBE.BAS on the Microsoft BASIC distribution disks.

```
' Define the macro string used to draw the cube
' and paint its sides:
One$ ="BR30 BU25 C1 R54 U45 L54 D45 BE20 P1,2 C2 G20"
Two$ ="R54 E20 L54 BD5 P2,2 U5 C4 G20 U45 E20 D45 BL5 P4,4"
Plot$ = One$ + Two$

APage% = 1      ' Initialize values for the active and visual
VPage% = 0      ' pages as well as the angle of rotation.
Angle% = 0
```

```
DO
   SCREEN 7, , APage%, VPage% ' Draw to the active page
                              ' while showing the visual page.

   CLS 1                      ' Clear the active page.

   ' Rotate the cube "Angle%" degrees:
   DRAW   "TA" + STR$(Angle%) + Plot$

   ' Angle% is some multiple of 15 degrees:
   Angle% = (Angle% + 15) MOD 360

   ' Drawing is complete, so make the cube visible in its
   ' new position by switching the active and visual pages:
   SWAP   APage%,VPage%

LOOP WHILE INKEY$ = ""        ' A keystroke ends the program.

END
```

Sample Applications

The sample applications in this chapter are a bar-graph generator, a program that plots points in the Mandelbrot Set using different colors, and a pattern editor.

Bar-Graph Generator (BAR.BAS)

This program uses all the forms of the **LINE** statement presented previously to draw a filled bar chart. Each bar is filled with a pattern specified in a **PAINT** statement. The input for the program consists of titles for the graph, labels for the x- and y-axes, and a set of up to five labels (with associated values) for the bars.

Statements Used

This program demonstrates the use of the following graphics statements:

- **LINE**
- **PAINT** (with a pattern)
- **SCREEN**

Program Listing

The bar-graph generator program BAR.BAS follows:

```
' Define type for the titles:
TYPE TitleType
   MainTitle AS STRING * 40
   XTitle AS STRING * 40
   YTitle AS STRING * 18
END TYPE

DECLARE SUB InputTitles (T AS TitleType)
DECLARE FUNCTION DrawGraph$ (T AS TitleType, Label$(), Value!(), N%)
DECLARE FUNCTION InputData% (Label$(), Value!())

' Variable declarations for titles and bar data:
DIM Titles AS TitleType, Label$(1 TO 5), Value(1 TO 5)

CONST FALSE = 0, TRUE = NOT FALSE

DO
   InputTitles Titles
   N% = InputData%(Label$(), Value())
   IF N% <> FALSE THEN
      NewGraph$ = DrawGraph$(Titles, Label$(), Value(), N%)
   END IF
LOOP WHILE NewGraph$ = "Y"

END

' ======================== DRAWGRAPH =====================
'   Draws a bar graph from the data entered in the
'   INPUTTITLES and INPUTDATA procedures.
' ========================================================

FUNCTION DrawGraph$ (T AS TitleType, Label$(), Value(), N%) STATIC

' Set size of graph:
CONST GRAPHTOP = 24, GRAPHBOTTOM = 171
CONST GRAPHLEFT = 48, GRAPHRIGHT = 624
CONST YLENGTH = GRAPHBOTTOM - GRAPHTOP

' Calculate maximum and minimum values:
YMax = 0
YMin = 0
FOR I% = 1 TO N%
   IF Value(I%) < YMin THEN YMin = Value(I%)
   IF Value(I%) > YMax THEN YMax = Value(I%)
NEXT I%
```

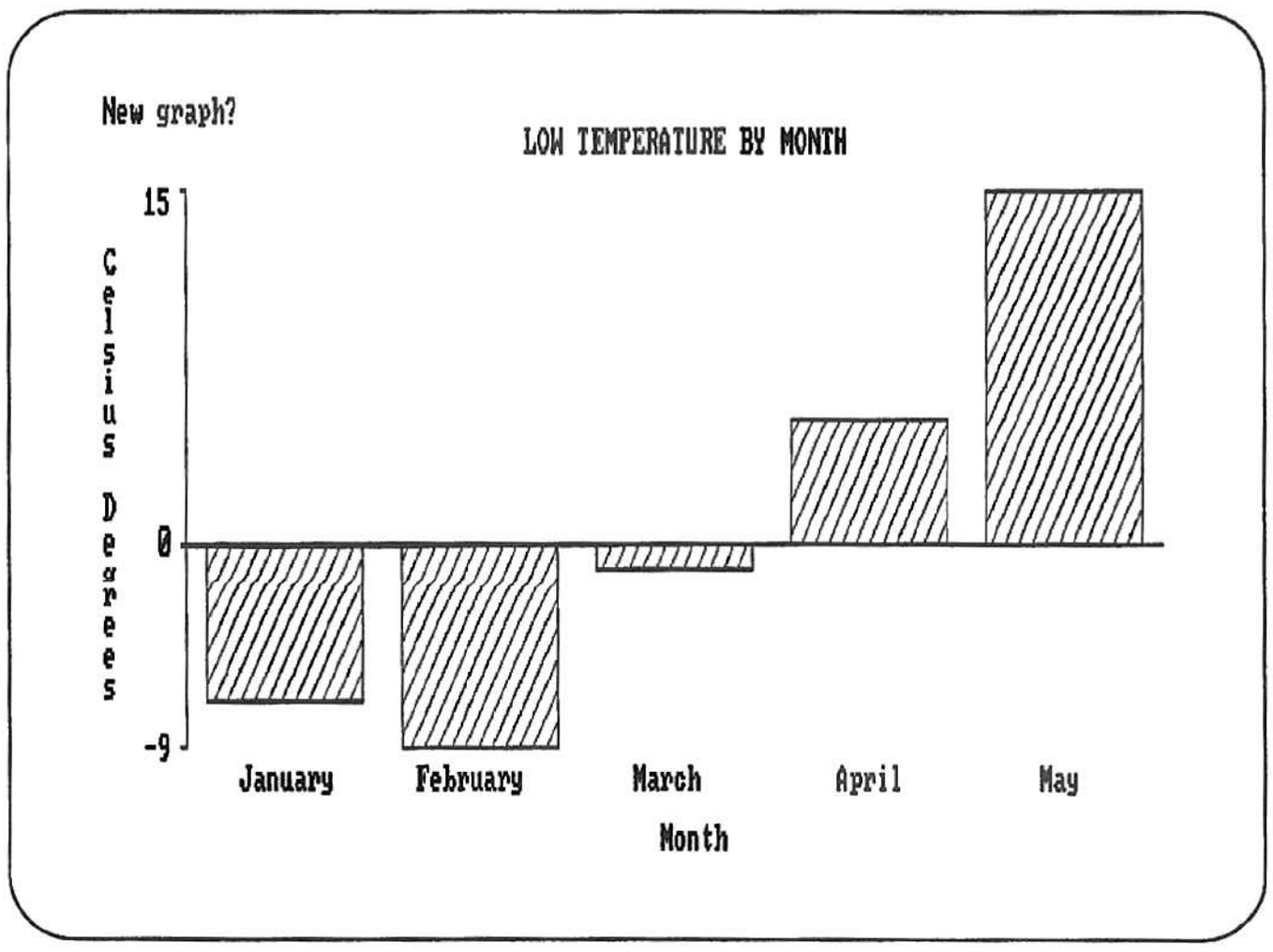
```
' Calculate width of bars and space between them:
BarWidth = (GRAPHRIGHT - GRAPHLEFT) / N%
BarSpace = .2 * BarWidth
BarWidth = BarWidth - BarSpace

SCREEN 2
CLS

' Draw y-axis:
LINE (GRAPHLEFT, GRAPHTOP)-(GRAPHLEFT, GRAPHBOTTOM), 1

' Draw main graph title:
Start% = 44 - (LEN(RTRIM$(T.MainTitle)) / 2)
LOCATE 2, Start%
PRINT RTRIM$(T.MainTitle);

' Annotate y-axis:
Start% = CINT(13 - LEN(RTRIM$(T.YTitle)) / 2)
FOR I% = 1 TO LEN(RTRIM$(T.YTitle))
   LOCATE Start% + I% - 1, 1
   PRINT MID$(T.YTitle, I%, 1);
NEXT I%

' Calculate scale factor so labels aren't bigger than four digits:
IF ABS(YMax) > ABS(YMin) THEN
   Power = YMax
ELSE
   Power = YMin
END IF
Power = CINT(LOG(ABS(Power) / 100) / LOG(10))
IF Power < 0 THEN Power = 0

' Scale minimum and maximum values down:
ScaleFactor = 10 ^ Power
YMax = CINT(YMax / ScaleFactor)
YMin = CINT(YMin / ScaleFactor)

' If power isn't zero then put scale factor on chart:
IF Power <> 0 THEN
   LOCATE 3, 2
   PRINT "x 10^"; LTRIM$(STR$(Power))
END IF

' Put tick mark and number for Max point on y-axis:
LINE (GRAPHLEFT - 3, GRAPHTOP) -STEP(3, 0)
LOCATE 4, 2
PRINT USING "####"; YMax
```

```
' Put tick mark and number for Min point on y-axis:
LINE (GRAPHLEFT - 3, GRAPHBOTTOM) -STEP(3, 0)
LOCATE 22, 2
PRINT USING "####"; YMin

YMax = YMax * ScaleFactor ' Scale minimum and maximum back
YMin = YMin * ScaleFactor ' up for charting calculations.

' Annotate x-axis:
Start% = 44 - (LEN(RTRIM$(T.XTitle)) / 2)
LOCATE 25, Start%
PRINT RTRIM$(T.XTitle);

' Calculate the pixel range for the y-axis:
YRange = YMax - YMin

' Define a diagonally striped pattern:
Tile$ = CHR$(1)+CHR$(2)+CHR$(4)+CHR$(8)+CHR$(16)+CHR$(32)+
        CHR$(64)+CHR$(128)

' Draw a zero line if appropriate:
IF YMin < 0 THEN
   Bottom = GRAPHBOTTOM - ((-YMin) / YRange * YLENGTH)
   LOCATE INT((Bottom - 1) / 8) + 1, 5
   PRINT "0";
ELSE
   Bottom = GRAPHBOTTOM
END IF

' Draw x-axis:
LINE (GRAPHLEFT - 3, Bottom)-(GRAPHRIGHT, Bottom)
' Draw bars and labels:
Start% = GRAPHLEFT + (BarSpace / 2)
FOR I% = 1 TO N%

   ' Draw a bar label:
   BarMid = Start% + (BarWidth / 2)
   CharMid = INT((BarMid - 1) / 8) + 1
   LOCATE 23, CharMid - INT(LEN(RTRIM$(Label$(I%))) / 2)
   PRINT Label$(I%);

   ' Draw the bar and fill it with the striped pattern:
   BarHeight = (Value(I%) / YRange) * YLENGTH
   LINE (Start%, Bottom) -STEP(BarWidth, -BarHeight), , B
   PAINT (BarMid, Bottom - (BarHeight / 2)), Tile$, 1

   Start% = Start% + BarWidth + BarSpace
NEXT I%
LOCATE 1, 1
PRINT "New graph? ";
DrawGraph$ = UCASE$(INPUT$(1))

END FUNCTION
```

```
' ========================= INPUTDATA =======================
'     Gets input for the bar labels and their values
' ===========================================================

FUNCTION InputData% (Label$(), Value()) STATIC

   ' Initialize the number of data values:
   NumData% = 0

   ' Print data-entry instructions:
   CLS
   PRINT "Enter data for up to 5 bars:"
   PRINT "   * Enter the label and value for each bar."
   PRINT "   * Values can be negative."
   PRINT "   * Enter a blank label to stop."
   PRINT
   PRINT "After viewing the graph, press any key ";
   PRINT "to end the program."

   ' Accept data until blank label or 5 entries:
   Done% = FALSE
   DO
      NumData% = NumData% + 1
      PRINT
      PRINT "Bar("; LTRIM$(STR$(NumData%)); "):"
      INPUT ; "        Label? ", Label$(NumData%)

      ' Only input value if label isn't blank:
      IF Label$(NumData%) <> "" THEN
    LOCATE , 35
    INPUT "Value? ", Value(NumData%)

      ' If label is blank, decrement data counter
      ' and set Done flag equal to TRUE:
      ELSE
    NumData% = NumData% - 1
     Done% = TRUE
        END IF
     LOOP UNTIL (NumData% = 5) OR Done%

     ' Return the number of data values input:
     InputData% = NumData%

END FUNCTION
```

```
' ====================== INPUTTITLES ======================
'     Accepts input for the three different graph titles
' =========================================================

SUB InputTitles (T AS TitleType) STATIC
   SCREEN 0, 0      ' Set text screen.
   DO         ' Input titles.
      CLS
      INPUT "Enter main graph title: ", T.MainTitle
      INPUT "Enter x-axis title    : ", T.XTitle
      INPUT "Enter y-axis title    : ", T.YTitle

      ' Check to see if titles are OK:
      LOCATE 7, 1
      PRINT "OK (Y to continue, N to change)? ";
      LOCATE , , 1
      OK$ = UCASE$(INPUT$(1))
   LOOP UNTIL OK$ = "Y"
END SUB
```

Output



Color in a Figure Generated Mathematically (MANDEL.BAS)

This program uses BASIC graphics statements to generate a figure known as a "fractal." A fractal is a graphic representation of what happens to numbers when they are subjected to a repeated sequence of mathematical operations. The fractal generated by this program shows a subset of the class of numbers known as complex numbers; this subset is called the "Mandelbrot Set," named after Benoit B. Mandelbrot of the IBM Thomas J. Watson Research Center.

Briefly, complex numbers have two parts, a real part and a so-called imaginary part, which is some multiple of $\sqrt{-1}$. Squaring a complex number, then plugging the real and imaginary parts back into a second complex number, squaring the new complex number, and repeating the process causes some complex numbers to get very large fairly fast. However, others hover around a stable value. The stable values are in the Mandelbrot Set and are represented in this program by the color black. The unstable values—that is, the ones that are moving away from the Mandelbrot Set—are represented by the other colors in the palette. The smaller the color attribute, the more unstable the point.

See A.K. Dewdney's column, "Computer Recreations," in *Scientific American*, August 1985, for more background on the Mandelbrot Set.

This program also tests for the presence of an EGA card, and if one is present, it draws the Mandelbrot Set in screen mode 8. After drawing each line, the program rotates the 16 colors in the palette with a **PALETTE USING** statement. If there is no EGA card, the program draws a four-color (white, magenta, cyan, and black) Mandelbrot Set in screen mode 1.

Statements and Functions Used

This program demonstrates the use of the following graphics statements and functions:

- **LINE**
- **PALETTE USING**
- **PMAP**
- **PSET**
- **SCREEN**
- **VIEW**
- **WINDOW**

Program Listing

```
DEFINT A-Z   ' Default variable type is integer.

DECLARE   SUB ShiftPalette ()
DECLARE   SUB WindowVals (WL%, WR%, WT%, WB%)
DECLARE   SUB ScreenTest (EM%, CR%, VL%, VR%, VT%, VB%)

CONST FALSE = 0, TRUE = NOT FALSE ' Boolean constants

' Set maximum number of iterations per point:
CONST MAXLOOP = 30, MAXSIZE = 1000000

DIM PaletteArray(15)
FOR I =   0 TO 15: PaletteArray(I) = I: NEXT I

' Call WindowVals to get coordinates of window corners:
WindowVals WLeft, WRight, WTop,    WBottom

' Call ScreenTest to find out if this is an EGA machine
' and get coordinates of viewport corners:
ScreenTest EgaMode, ColorRange,    VLeft, VRight, VTop, VBottom

' Define viewport and corresponding window:
VIEW (VLeft, VTop)-(VRight, VBottom), 0, ColorRange
WINDOW (WLeft, WTop)-(WRight, WBottom)

LOCATE 24, 10 : PRINT "Press any key to quit.";

XLength   = VRight - VLeft
YLength   = VBottom - VTop
ColorWidth = MAXLOOP \ ColorRange

' Loop through each pixel in viewport and calculate
' whether or not it is in the Mandelbrot Set:
FOR Y =   0 TO YLength          ' Loop through every line
                                ' in the viewport.
   LogicY = PMAP(Y, 3)          ' Get the pixel's window
                                ' y-coordinate.
   PSET   (WLeft,LogicY)        ' Plot leftmost pixel in the line.
   OldColor = 0                 ' Start with background color.

   FOR X = 0 TO XLength         ' Loop through every pixel
                                ' in the line.
      LogicX = PMAP(X, 2)       ' Get the pixel's window
                                ' x-coordinate.
      MandelX& = LogicX
      MandelY& = LogicY
```

```
' Do the calculations to see if this point
' is in the Mandelbrot Set:
FOR I = 1 TO MAXLOOP
   RealNum& = MandelX& * MandelX&
   ImagNum& = MandelY& * MandelY&
   IF (RealNum& + ImagNum&) >= MAXSIZE THEN EXIT FOR
   MandelY& = (MandelX& * MandelY&) \ 250 + LogicY
   MandelX& = (RealNum& - ImagNum&) \ 500 + LogicX
NEXT I

' Assign a color to the point:
PColor = I \ ColorWidth

' If color has changed, draw a line from
   ' the last point referenced to the new point,
   ' using the old color:
   IF PColor <> OldColor THEN
 LINE -(LogicX, LogicY), (ColorRange - OldColor)
 OldColor = PColor
   END IF

   IF INKEY$ <> "" THEN END
NEXT   X

' Draw the last line segment to the right edge
' of the viewport:
LINE   -(LogicX, LogicY), (ColorRange - OldColor)

' If this is an EGA machine, shift the palette after
' drawing each line:
IF EgaMode THEN ShiftPalette
NEXT Y

DO
   ' Continue shifting the palette
   ' until the user presses a key:
   IF EgaMode THEN ShiftPalette
LOOP WHILE INKEY$ = ""

SCREEN 0, 0              ' Restore the screen to text mode,
WIDTH 80                 ' 80 columns.
END

BadScreen:               ' Error handler that is invoked if
   EgaMode = FALSE       ' there is no EGA graphics card.
   RESUME NEXT
```

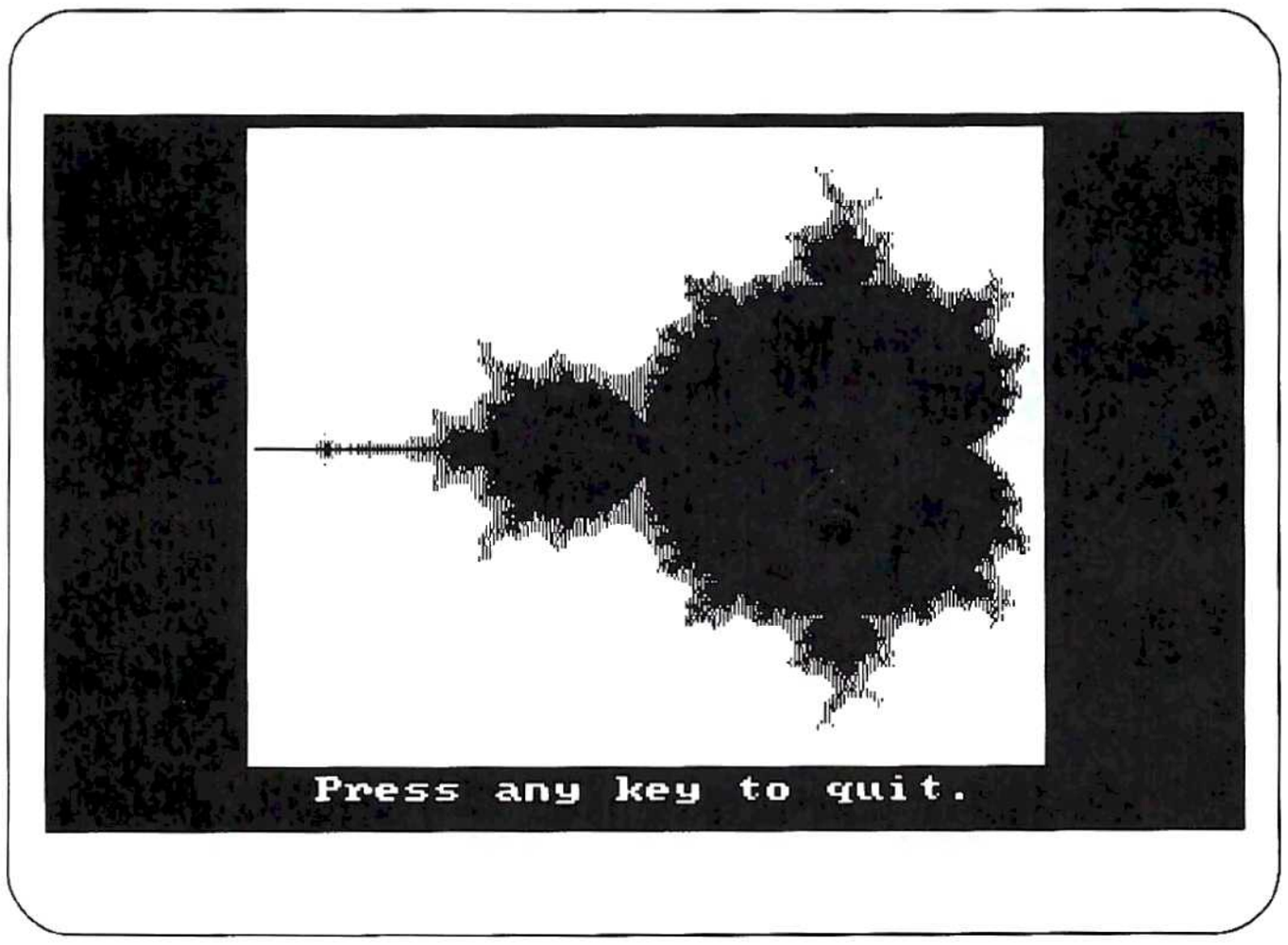
```
' ====================== ShiftPalette ====================
'    Rotates the palette by one each time it is called
' ========================================================

SUB ShiftPalette STATIC
   SHARED PaletteArray(), ColorRange

   FOR I = 1 TO ColorRange
      PaletteArray(I) =(PaletteArray(I) MOD ColorRange) + 1
   NEXT   I
   PALETTE USING PaletteArray(0)

END SUB
```

```
' ====================== ScreenTest ======================
'    Uses a SCREEN 8 statement as a test to see if user has
'    EGA hardware. If this causes an error, the EM flag is
'    set to FALSE, and the screen is set with SCREEN 1.

'    Also sets values for corners of viewport (VL = left,
'    VR = right, VT = top, VB = bottom), scaled with the
'    correct aspect ratio so viewport is a perfect square.
' ========================================================

SUB ScreenTest (EM, CR,VL, VR,      VT, VB) STATIC
   EM =   TRUE
   ON ERROR GOTO BadScreen
   SCREEN 8, 1
   ON ERROR GOTO 0

   IF EM THEN           ' No error, SCREEN 8 is OK.
      VL = 110: VR = 529
      VT = 5: VB = 179
      CR = 15           ' 16 colors (0 - 15)

   ELSE                 ' Error, so use SCREEN 1.
      SCREEN 1, 1
      VL = 55: VR = 264
      VT = 5: VB = 179
      CR = 3            ' 4 colors (0 - 3)
   END IF

END SUB
```

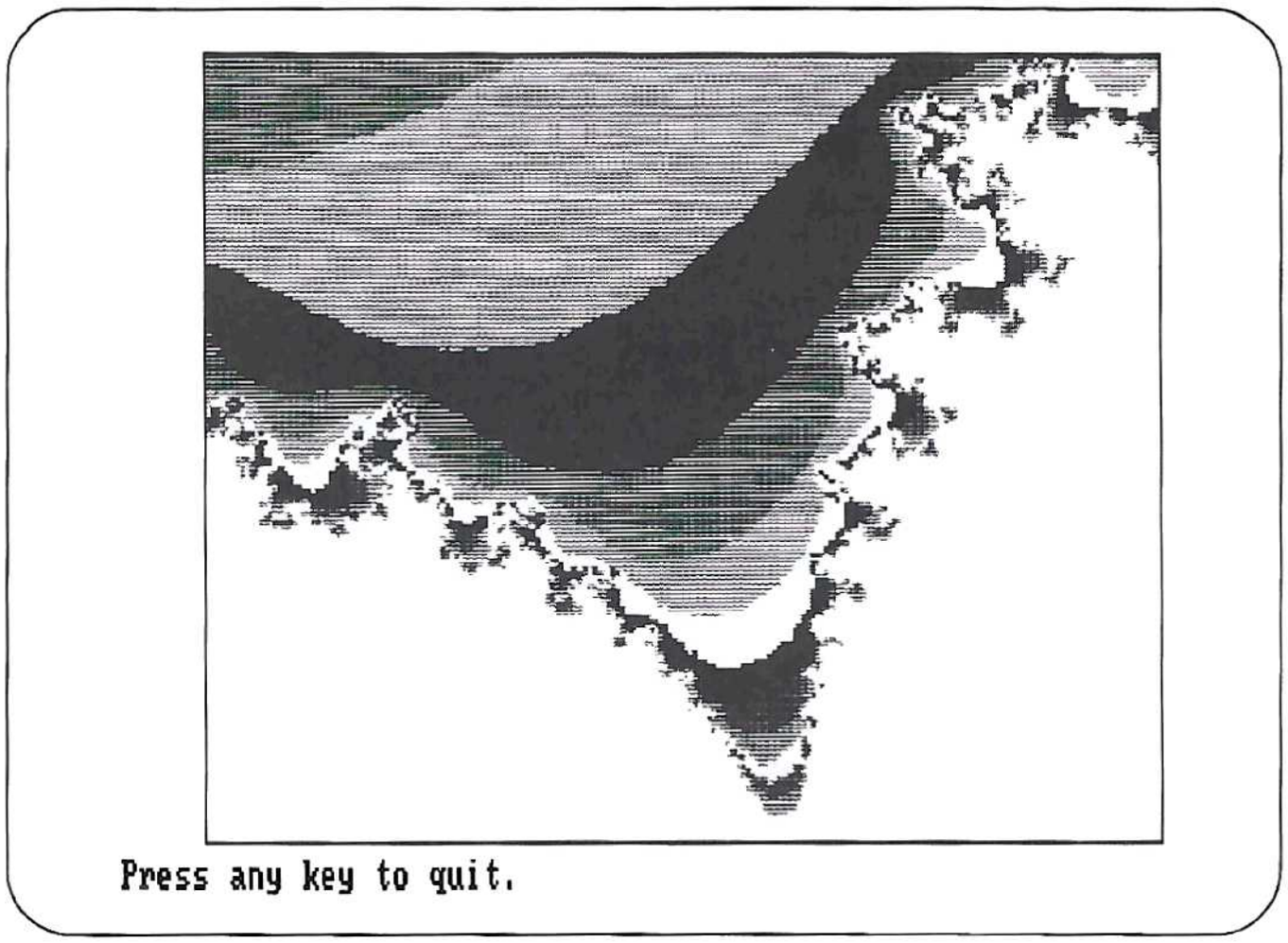
```
' ======================= WindowVals ======================
'     Gets window corners as input from the user, or sets
'     values for the corners if there is no input.
' =========================================================

SUB WindowVals (WL, WR,WT, WB)       STATIC
   CLS
   PRINT "This program prints the graphic representation of"
   PRINT "the complete Mandelbrot Set. The default window"
   PRINT "is from (-1000,625) to (250,-625). To zoom in on"
   PRINT "part of the figure, input coordinates inside"
   PRINT "this window."
   PRINT "Press <ENTER> to see the default window or"
   PRINT "any other key to input window coordinates: ";
   LOCATE , , 1
   Resp$ = INPUT$(1)

   ' User didn't press ENTER, so input window corners:
   IF Resp$ <> CHR$(13)THEN
      PRINT
      INPUT "x-coordinate of upper-left corner: ", WL
      DO
    INPUT "x-coordinate of lower-right corner: ", WR
    IF WR <= WL THEN
       PRINT "Right corner must be greater than left corner."
    END IF
      LOOP WHILE WR <= WL
      INPUT "y-coordinate of upper-left corner: ", WT
      DO
    INPUT "y-coordinate of lower-right corner: ", WB
    IF WB >= WT THEN
       PRINT "Bottom corner must be less than top corner."
    END IF
      LOOP WHILE WB >= WT

   ' User pressed Enter, so set default values:
      ELSE
         WL = -1000
         WR = 250
         WT = 625
         WB = -625
      END IF
   END SUB
```

Output

The following figure shows the Mandelbrot Set in screen mode 1. This is the output you see if you have a CGA and you choose the default window coordinates.



The next figure shows the Mandelbrot Set with (*x*, *y*) coordinates of (–500, 250) for the upper-left corner and (–300, 50) for the lower-right corner. This figure is drawn in screen mode 8, the default for an EGA or VGA.



Pattern Editor (EDPAT.BAS)

This program allows you to edit a pattern tile for use with **PAINT**. While you are editing the tile on the left side of the screen, you can check the appearance of the finished pattern on the right side of the screen. When you have finished editing the pattern tile, the program prints the integer arguments used by the **CHR$** function to draw each row of the tile.

Statements Used

This program demonstrates the use of the following graphics statements:

- **LINE**
- **PAINT** (with pattern)
- **VIEW**

Program Listing

```
DECLARE   SUB DrawPattern ()
DECLARE   SUB EditPattern ()
DECLARE   SUB Initialize ()
DECLARE   SUB ShowPattern (OK$)

DIM Bit%(0 TO 7), Pattern$, Esc$, PatternSize%

DO
   Initialize
   EditPattern
   ShowPattern OK$
LOOP WHILE OK$ = "Y"
END

' ======================== DRAWPATTERN ====================
'  Draws a patterned rectangle on the right side of screen.
' =========================================================

SUB DrawPattern STATIC
SHARED Pattern$
   VIEW   (320, 24)-(622, 160), 0, 1  ' Set view to rectangle.
   PAINT (1, 1), Pattern$                  ' Use PAINT to fill it.
   VIEW                                    ' Set view to full screen.

END SUB

' ======================== EDITPATTERN ====================
'                    Edits a tile-byte pattern.
' =========================================================

SUB EditPattern STATIC
SHARED Pattern$, Esc$, Bit%(), PatternSize%
ByteNum% = 1     ' Starting position.
BitNum% = 7
Null$ = CHR$(0)  ' CHR$(0) is the first byte of the
      ' two-byte string returned when a
      ' direction key such as Up or Down is
      ' pressed.
DO

   '   Calculate starting location on screen of this bit:
   X% = ((7 - BitNum%) * 16)       + 80
   Y% = (ByteNum% + 2) * 8
```

```
    '   Wait for a key press (flash cursor each 3/10 second):
    State% = 0
    RefTime = 0
    DO

  ' Check timer and switch cursor state if 3/10 second:
  IF ABS(TIMER - RefTime) > .3 THEN
     RefTime = TIMER
     State% = 1 - State%

     ' Turn the border of bit on and off:
     LINE (X%-1, Y%-1) -STEP(15, 8), State%, B
  END IF

  Check$ = INKEY$                          ' Check for keystroke.

     LOOP WHILE Check$= ""                 ' Loop until a key is pressed.

     '   Erase cursor:
     LINE (X%-1, Y%-1) -STEP(15, 8), 0, B

     SELECT CASE Check$                    ' Respond to keystroke.

     CASE CHR$(27)                              ' Esc key pressed:
        EXIT SUB                           ' exit this subprogram.
  CASE CHR$(32)                            ' Spacebar pressed:
                                           ' reset state of bit.

        ' Invert bit in pattern string:
        CurrentByte% = ASC(MID$(Pattern$, ByteNum%, 1))
        CurrentByte% = CurrentByte% XOR Bit%(BitNum%)
        MID$ (Pattern$, ByteNum%) = CHR$(CurrentByte%)

        ' Redraw bit on screen:
        IF (CurrentByte% AND Bit%(BitNum%)) <> 0 THEN
            CurrentColor% = 1
        ELSE
            CurrentColor% = 0
        END IF
        LINE (X%+1, Y%+1) -STEP(11, 4), CurrentColor%, BF

     CASE CHR$(13)                         ' Enter key pressed: draw
        DrawPattern                        ' pattern in box on right.

     CASE Null$ + CHR$(75)                 ' Left key: move cursor left.

        BitNum% = BitNum% + 1
        IF  BitNum%> 7 THEN BitNum% = 0

     CASE Null$ + CHR$(77)                 ' Right key: move cursor right.
```

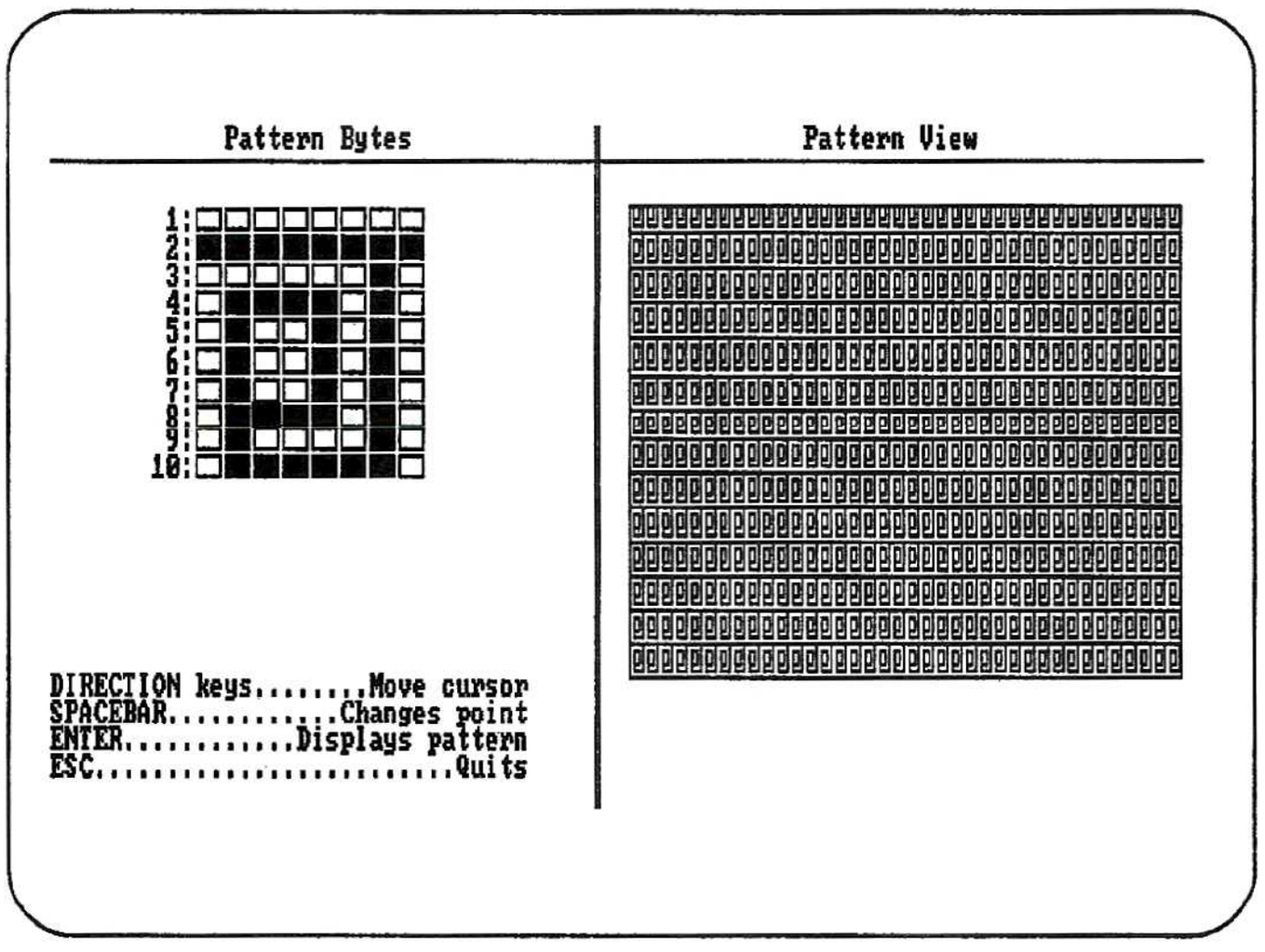
```
      BitNum% = BitNum% - 1
      IF  BitNum%< 0 THEN BitNum% = 7

   CASE Null$ + CHR$(72)          ' Up key: move cursor up.

      ByteNum% =ByteNum% - 1
      IF  ByteNum% < 1 THEN ByteNum% = PatternSize%

   CASE Null$ + CHR$(80)          ' Down key: move cursor down.

      ByteNum% =ByteNum% + 1
      IF  ByteNum% > PatternSize% THEN ByteNum% = 1

   CASE ELSE
      ' User pressed a key other than Esc, Spacebar,
      ' Enter, Up, Down, Left, or Right, so don't
      ' do anything.
      END SELECT
   LOOP
END SUB

' ======================== INITIALIZE ========================
'             Sets up starting pattern and screen
' ============================================================

SUB Initialize STATIC
SHARED Pattern$, Esc$, Bit%(), PatternSize%

   Esc$   = CHR$(27)    ' Esc character is ASCII 27.

   ' Set up an array holding bits in positions 0 to 7:
   FOR I% = 0 TO 7
      Bit%(I%) = 2 ^ I%
   NEXT  I%

   CLS

   ' Input the pattern size (in number of bytes):
   LOCATE 5, 5
   PRINT "Enter pattern size (1-16 rows):";
   DO
      LOCATE 5, 38
      PRINT " ";
      LOCATE 5, 38
      INPUT "", PatternSize%
   LOOP  WHILE PatternSize% < 1 OR PatternSize% > 16
```

```
    ' Set initial pattern to all bits set:
    Pattern$ = STRING$(PatternSize%, 255)

    SCREEN 2      ' 640 x 200 monochrome graphics mode

    ' Draw dividing lines:
    LINE   (0, 10)-(635, 10), 1
    LINE   (300, 0)-(300, 199)
    LINE   (302, 0)-(302, 199)

    ' Print titles:
    LOCATE 1, 13: PRINT "Pattern Bytes"
    LOCATE 1, 53: PRINT "Pattern View"

' Draw editing screen for pattern:
FOR I% = 1 TO PatternSize%

   ' Print label on left of each line:
   LOCATE I% + 3, 8
   PRINT USING "##:"; I%

   ' Draw "bit" boxes:
   X% = 80
   Y% = (I% + 2) * 8
   FOR J% = 1 TO 8
 LINE (X%, Y%) -STEP(13, 6), 1,   BF
 X% = X% + 16
   NEXT J%
NEXT   I%

DrawPattern      ' Draw "Pattern View" box.

LOCATE 21, 1
PRINT "DIRECTION keys........Move cursor"
PRINT "SPACEBAR.............Changes point"
PRINT "ENTER................Displays pattern"
   PRINT "ESC............................Quits";

END SUB
```

```
' ========================= SHOWPATTERN ====================
'   Prints the CHR$ values used by PAINT to make pattern
' ==========================================================

SUB ShowPattern (OK$) STATIC
SHARED Pattern$, PatternSize%

   ' Return screen to 80-column text mode:
   SCREEN 0, 0
   WIDTH 80

   PRINT "The following characters make up your pattern:"
   PRINT

   ' Print out the value for each pattern byte:
   FOR I% = 1 TO PatternSize%
      PatternByte% = ASC(MID$(Pattern$,    I%, 1))
      PRINT "CHR$("; LTRIM$(STR$(PatternByte%)); ")"
   NEXT   I%
   PRINT
   LOCATE , , 1
   PRINT "New pattern? ";
   OK$ = UCASE$(INPUT$(1))
END SUB
```

Output

This is a sample pattern generated by the pattern editor.



Chapter 6
Presentation Graphics

Microsoft BASIC includes a toolbox of BASIC **SUB** and **FUNCTION** procedures, and assembly language routines you can use to add charts and graphs to your programs quickly and easily. These procedures are collectively known as the Presentation Graphics toolbox and include support for pie charts, bar and column charts, line graphs, and scatter diagrams. Each of these types of charts can convert masses of numbers to a single expressive picture.

This chapter shows you how to use the Presentation Graphics toolbox in your BASIC programs. The first section describes which files you need to use the Presentation Graphics toolbox and demonstrates how it simplifies the graphic presentation of data. Subsequent sections explain terminology, present more elaborate examples, and describe some of the toolbox's many capabilities.

You'll also learn about Presentation Graphics' default data structures and how to manipulate them. The final section presents a short reference list of all the routines that comprise the Presentation Graphics toolbox and shows you how to include custom graphics fonts in your charts.

To use the Presentation Graphics toolbox you need a graphics adapter and a monitor capable of bit-mapped display—the same equipment mentioned in Chapter 5, "Graphics." Support is provided for CGA, EGA, VGA, MCGA, Hercules monochrome graphics, and the Olivetti Color Board.

The BASIC procedures for Presentation Graphics are contained in the source-code module CHRTB.BAS. The assembly language routines are in the object file CHRTASM.OBJ. When you ran the Setup program, you had an opportunity to have a Quick library (.QLB) and a object-module libraries (.LIB) created that contain all necessary BASIC and assembly language routines. To write presentation graphics programs within the QBX environment, load the Quick library CHRTBEFR.QLB when you start QBX, for example:

```
QBX /L CHRTBEFR
```

Table 6.1 lists files and libraries that relate to Presentation Graphics toolbox. The "???" stands for the combination of characteristics you chose for your object files and libraries during Setup. E or A in the first position corresponds to your choice of emulator math or alternate math; F or N in the second position corresponds to your choice of near or far strings; R or P in the third position corresponds to your choice of target environments for executable files, either real or protected mode. For instance, CHRTBEFR.LIB is a library that uses the emulator math package, far strings, and runs only in DOS or the real-mode "compatibility box" of OS/2.

Table 6.1 Presentation Graphics Toolbox Files

Source file	.OBJ file	.QLB file	.LIB file
CHRTB.BAS	CHRTB???.OBJ	CHRTBEFR.QLB	CHRTB???.LIB
CHRTB.BI			
Not supplied	CHRTASM.OBJ	CHRTBEFR.QLB	CHRTB???.LIB
FONTB.BAS	FONTB???.OBJ	FONTBEFR.QLB and CHRTBEFR.QLB	FONTB???.LIB
FONTB.BI			
Not supplied	FONTASM.OBJ	FONTBEFR.QLB and CHRTBEFR.QLB	FONTB???.LIB

Quick libraries can only use the far strings, emulator, and real mode options because these are the only possibilities in the QBX environment. You can construct object-module libraries with the near-strings and alternate math options, but no BASIC graphics programs can run under OS/2. If you want your executable files to use near strings and/or alternate math, you must compile them from the command line because the Alternate Math, Near Strings and OS/2 Protected Mode options are disabled within QBX whenever a Quick library is loaded.

The .QLB and .LIB files were created in the \BC7\LIB directory of your root directory, unless you specified a different directory during Setup. Check the file PACKING.LST on your distribution disks for further information on the .BAS and .OBJ files and to find out where they appear on the distribution disks.

When you create a stand-alone executable program from within the QBX environment from code that uses Presentation Graphics toolbox, the appropriate Presentation Graphics toolbox object-module library must be in the directory specified in the Option menu's Set Paths dialog box. Using libraries is discussed in Chapter 18, "Using LINK and LIB."

The graphic fonts procedures represented by FONTB.BAS, FONTASM.OBJ, and FONTB.BI are built into CHRTBEFR.QLB and other CHRTB object-module libraries during Setup. You can use the graphic fonts in your Presentation Graphics toolbox programs without loading anything but CHRTBEFR.QLB when you start QBX. The graphic fonts are also built as a completely separate library (FONTBEFR.QLB). You can use the graphic fonts separately and combine them with any other libraries. Two font files (TMSRB.FON and HELVB.FON) are supplied, but you can use any Microsoft Windows compatible bitmap fonts with these procedures. These files are designed specifically for the EGA aspect ratio.

Presentation Graphics Program Structure

Typically in programming, you have to have a thorough understanding of a tool before you can even begin using it. However, with the Presentation Graphics toolbox you have a choice about the level of understanding you wish to cultivate. Even if you don't understand the Presentation Graphics toolbox any better than you do right now, you can add very adequate charts to your programs by following a few simple steps. To create bar, column, and pie charts, you can simply collect data, label the data's parts, and then call three routines to chart it on the screen. The following example demonstrates how simple it is to make the chart shown in Figure 6.1 with Presentation Graphics toolbox.

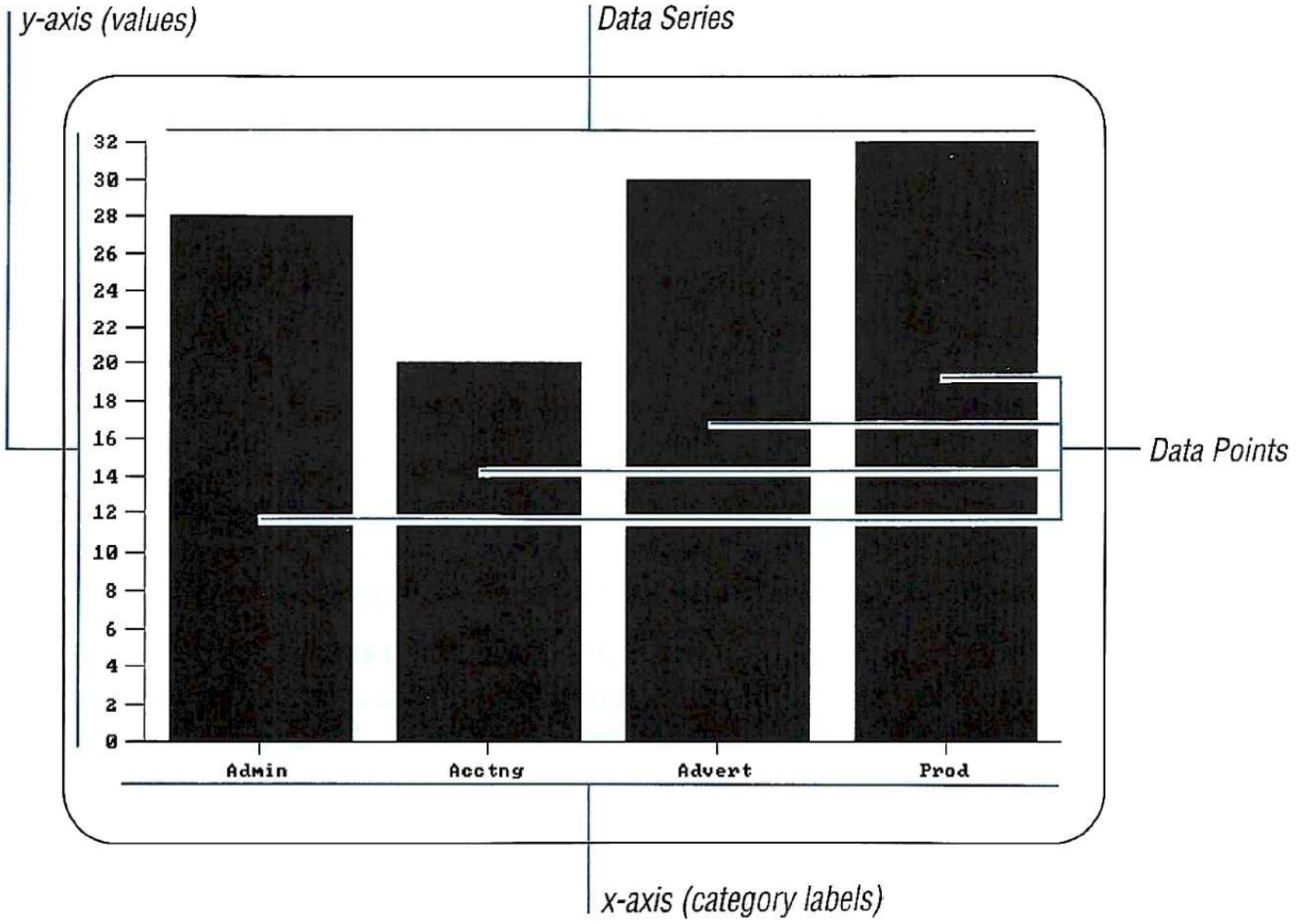


Figure 6.1 Column Chart with Characteristics Labelled

1. Specify the proper include file:

```
' $INCLUDE: 'CHRTB.BI'
```

   You must specify the file CHRTB.BI to call Presentation Graphics toolbox routines.

2. Declare variables to pass as arguments to the presentation graphics routines:

```
DIM Env AS ChartEnvironment
DIM DataValues(1 TO 4) AS SINGLE
DIM Labels(1 TO 4) AS STRING
```

You need a variable of the user-defined type ChartEnvironment, plus data and the labels for the charted array variables for your charted data. These arrays are always dimensioned starting at 1 (rather than 0). Numeric data values are always single precision.

3. Assemble the plot data (the following **DATA** and **READ** statements simulate data collection):

```
DATA 28, 20, 30, 32
DATA "Admin", "Acctng", "Advert", "Prod"
FOR I=1 TO 4
    READ DataValues(I)
NEXT I
FOR I=1 TO 4
    READ Labels(I)
NEXT I
```

Data can come from a variety of sources. It can result from processing elsewhere in the program, be read from files, or even entered from the keyboard. Wherever it comes from, you must place it in the arrays dimensioned in the preceding step.

4. Use the Presentation Graphics toolbox routine **ChartScreen** to set the video mode. You cannot use the **SCREEN** statement:

```
ChartScreen 2
```

You use the **ChartScreen** routine as you would normally use the **SCREEN** statement. In this case 2 is passed because it gives graphics on a common hardware setup (CGA). If you have a Hercules (or compatible) setup, pass a 3 to **ChartScreen**; if you have an Olivetti, pass a 4. Later examples illustrate how to choose the best mode for your user's hardware.

5. Use the Presentation Graphics toolbox routine **DefaultChart** to set up the chart environment to display the type of chart you want. You pass the ChartEnvironment type variable declared in step 2, plus constants that describe the kind of chart and its features:

```
DefaultChart Env, cColumn, cPlain
```

DefaultChart supplies most of the settings you want. After defaults are set with **DefaultChart**, you can modify them with simple assignment statements (as illustrated in the examples later in this chapter).

6. Use the appropriate Presentation Graphics toolbox routine to display the chart. You pass the variables declared in step 2, plus an integer that specifies the number of values:

```
Chart Env, Labels(), DataValues(), 4
```

There are separate routines for standard charts (i.e. column, bar, and line charts), for pie charts, scatter charts, and charts that display multiple data series.

7. Pause execution while chart is displayed:

```
SLEEP
```

You can use BASIC's new **SLEEP** statement to keep the chart on the screen for viewing.

8. Reset the video mode (optional):

```
SCREEN 0
```

When your program detects the signal to continue, you may want to reset the video mode if graphics are not necessary for the rest of the program.

Terminology

The following explanations of the terms and phrases used when discussing the Presentation Graphics toolbox will help you better understand this chapter and its contents.

Data Point

A data item having a numeric value. In a chart, a data point is usually one value among a series of values to be illustrated. Data points are shown either as bars, columns, slices of a pie, or as individual plot characters (markers). In the "Presentation Graphics Program Structure" section, each of the elements of the `DataValues()` array was a data point.

Data Series

Groups or series of data that can be graphed on the same chart, for example, as a continuous set of data points on a graph. In the preceding section, the collection of elements comprising the `DataValues()` array represented a data series.

Data items related by a common idea or purpose constitute a "series." For example, the day-to-day prices of a stock over the course of a year form a single data series. The Presentation Graphics toolbox allows you to plot multiple series on the same graph. In theory only your system's memory capacity restricts the number of data series that can appear on a graph. However, there are practical considerations.

Characteristics such as color and pattern help distinguish one series from another; you can more readily differentiate series on a color monitor than you can on a monochrome monitor. The number of series that can comfortably appear on the same chart depends on the chart type and the number of available colors. Only experimentation can tell you what is best for the system on which your program will run.

Categories

"Categories" are non-numeric data. A set of categories forms a frame of reference for comparisons of numeric data. For example, the months of the year are categories against which numeric data such as rainfall can be plotted. In the example in the section "Presentation Graphics Program Structure," each element of the `Labels()` array was a category.

Regional sales provide another example. A chart can show comparisons of a company's sales in different parts of the country. Each region forms a category. The sales within each region are numeric data that have meaning only within the context of a particular category.

Values

"Values" are numeric data. Each value could be represented on a chart by a data point. Each element (or data point) in a data series has a value. Sales, stock prices, air temperatures, populations—all are series of values that can be plotted against categories or against other values.

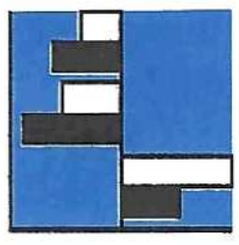
The Presentation Graphics toolbox allows you to represent different series of value data on a single graph. For example, average monthly temperatures or monthly sales of heating oil during different years—or a combination of temperatures and sales—can be plotted together on the same graph.

Pie Charts

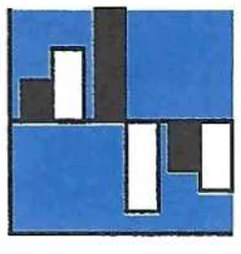
"Pie charts" illustrate the relationship between each element of a data series and the whole series. A good example is a company's sales to various accounts. Each account can be represented as a slice of the pie.

Presentation graphics can display either a standard or an "exploded" pie chart. The exploded view shows the pie with one or more pieces separated out for emphasis. Presentation graphics optionally labels each slice of a pie chart with a percentage figure. You use a legend to associate each slice of the pie with a category name.

Bar and Column Charts

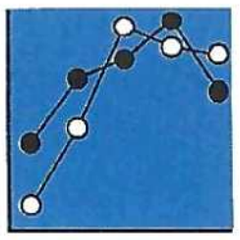


As the name implies, a "bar chart" shows data as horizontal bars. Simple bar charts show the absolute value of each category.



"Column charts" represent data as vertical columns. Column charts are frequently used to show variations over a period of time, since they illustrate time flow better than bar charts.

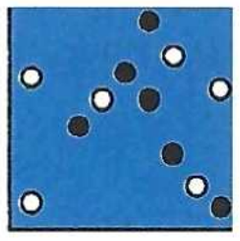
Line Charts



"Line charts" illustrate trends or changes in data. They show how a series of values varies against some category—for example, average temperatures throughout a particular year.

Traditionally, line charts show a collection of data points connected by lines; hence the name. However, Presentation Graphics toolbox can also plot points that are not connected by lines.

Scatter Diagrams



"Scatter diagrams" display values against other values. A scatter diagram simply plots points in an x-y coordinate grid.

Scatter diagrams illustrate the relationship between numeric values in different groups of data to show trends and correlations. This is why scatter diagrams are a favorite tool of statisticians and forecasters.

They are most useful with relatively large populations of data. Consider, for example, the relationship between personal income and family size. If you poll 1,000 wage earners for their income and family size, you have a scatter diagram with 1,000 points. If you combine your results so you're left with one average income for each family size, you have a line graph.

Sometimes the related points on scatter charts are connected by lines. For example, if you plotted the mathematical relationship $f(x)=x^2$, you would plot the value x against the value x^2, and connecting the points with lines would illustrate the relationship. However, for statistical graphs involving large groups, connecting values with lines would make the chart incomprehensible.

Axes

All charts created with the Presentation Graphics toolbox (except pie charts) are displayed with two perpendicular reference lines called "axes." These axes are "yardsticks" against which data is charted. Generally, the vertical or y-axis runs from top to bottom of the chart and is placed against the left side of the screen. The horizontal or x-axis runs from left to right across the bottom of the screen.

The chart type determines which axis is used for category data and which axis is used for value data. The x-axis is the category axis for column and line charts and the value axis for bar charts. The y-axis is the value axis for column and line charts and the category axis for bar charts. The x- and y-axes are used for value data in scatter charts.

Chart Windows

The "chart window" defines that part of the screen on which the chart is drawn. Normally the window fills the entire screen, but Presentation Graphics toolbox allows you to resize the window for smaller graphs. By moving the chart window to different screen locations, you can view separate graphs together on the same screen.

Data Windows

While the chart window defines the entire graph including axes and labels, the "data window" defines only the actual plotting area. This is the portion of the graph to the right of the y-axis and above the x-axis. You cannot directly specify the size of the data window. Presentation Graphics automatically determines its size based on the dimensions of the chart window.

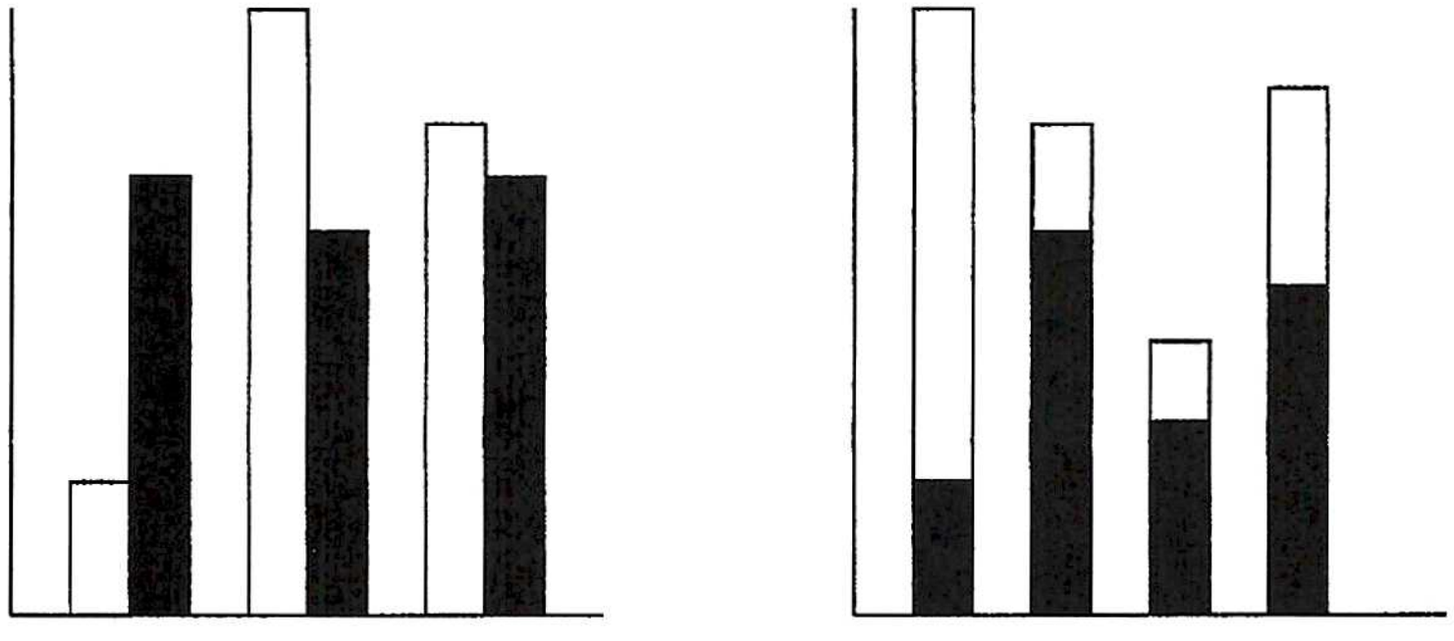
Chart Styles

Each of the five types of Presentation Graphics toolbox charts can appear in two different "chart styles," as described in Table 6.2.

Table 6.2 Presentation Graphics Chart Styles

Chart type	Chart style #1	Chart style #2
Pie	With percentages	Without percentages
Bar	Side-by-side	Stacked
Column	Side-by-side	Stacked
Line	Points connected by lines	Points only
Scatter	Points connected by lines	Points only

Bar and column charts have only one style when displaying a single series of data. The styles "side-by-side" and "stacked" are applicable when more than one series appears on the same chart. The first style arranges the bars or columns for the different series side by side, showing relative heights or lengths. The stacked style, illustrated in Figure 6.2 for a column chart, emphasizes relative sizes between bars or columns and shows the totals of the series.



Side-By-Side Column

Stacked Column

Figure 6.2 Side-by-Side and Stacked Styles for Typical Column Chart

Legends

Presentation graphics can display a "legend" to label the different series of a chart, in addition to differentiating between series by using colors, lines, or patterns. Pie charts, which only represent a single series, can use a legend to identify each "slice" of the pie.

The format of a legend is similar to the legends found on printed graphs and maps. A sample of the color and pattern used to graph each distinct data series appears next to the series label. The section "Palettes" later in this chapter, explains how different data series are identified by color and pattern.

Five Example Chart Programs

The sample programs that follow use only five of the 19 procedures in the Presentation Graphics toolbox: **DefaultChart**, **ChartScreen**, **ChartPie**, **Chart**, and **ChartScatter**. The *BASIC Language Reference* describes these and the remaining Presentation Graphics toolbox. For information on including online Help for presentation graphics routines in the Microsoft Advisor online Help system, see Chapter 22, "Customizing Online Help."

The code in the example programs is straightforward, and you should be able to follow the programs easily without completely understanding all the details. Each program is described with comments so that you can recognize the steps mentioned earlier in the section "Presentation Graphics Program Structure." The examples make use of the same secondary procedure, `BestMode`. `BestMode` checks the display adapter and returns the best available mode value for displaying charts. As in the preceding example, **DATA** and **READ** statements are used to simulate data generation.

A Sample Data Set

Suppose a grocer wants to graph the sales of orange juice over the course of a single year. Sales figures are on a monthly basis, so the grocer selects as category data the months of the year from January through December. The sales figures are shown in Table 6.3.

Table 6.3 Good Neighbor Grocery Orange Juice Sales for Year

Month	Quantity (cases)
January	33
February	27
March	42
April	64
May	106
June	157
July	182
August	217
September	128
October	62
November	43
December	36

Example: Pie Chart

The example in this section uses Presentation Graphics toolbox to display a pie chart for the grocer's data. Interesting elements of the pie-charting example include:

- **Exploded slices** A feature unique to pie charts is that any or all of the slices can be separated from the rest of the chart for emphasis. When pieces are separated, they are sometimes referred to as "exploded." You designate which slices should be separated from the rest by defining an integer array of "flags," then setting the flags by assigning non-zero values to those array elements corresponding to the pie slices you want to have separated from the rest. In the example in this section, the `Exploded()` array causes any slice of the pie representing a peak sales month (i.e., `OJvalues` value greater than or equal to 100) to be to be separated from the rest.
- **Setting chart characteristics** The wording, alignment, and color of the chart's main title and subtitle are set by assigning values to elements of the structured variable `Env` (a variable of user-defined type ChartEnvironment). A constant assigned to another element of the `Env` variable specifies that the chart itself is to have no border (`Env.Chartwindow.Border = cNo`). Note that in all these cases, the elements themselves are of user-defined type.

The section "Customizing Presentation Graphics" later in this chapter describes the ChartEnvironment type, as well as its constituent types, including TitleType (for the elements `MainTitle.Title`, `MainTitle.TitleColor`, `MainTitle.Justify`, `SubTitle.Title`, `SubTitle.TitleColor`, and `SubTitle.Justify`) and RegionType (for the `ChartWindow.Border` element). Because of the nesting of user-defined types, the names of these variables can be lengthy, but the principle is a simple one: specifying the appearance of any chart by simple assignment of values to variables that are the same for every chart.

- **Presentation graphics error codes** When an error occurs during execution of a Presentation Graphics toolbox routine, an error condition code is placed in the `ChartErr` variable (defined in the common block `/ChartLib/`). A `ChartErr` value of 0 indicates the routine has completed its work without error. In the following examples, `ChartErr` is checked after the call to the **ChartScreen** routine to make sure the chart can be displayed. See the include file CHRTB.BI for a listing of the error codes.

Example The following example displays a pie chart (PGPIE.BAS) based on the values in Table 6.3.

```
' PGPIE.BAS: Create sample pie chart

DEFINT A-Z
' $INCLUDE: 'FONTB.BI'
' $INCLUDE: 'CHRTB.BI'
DECLARE FUNCTION BestMode ()
CONST FALSE = 0, TRUE = NOT FALSE, MONTHS = 12
CONST HIGHESTMODE = 13, TEXTONLY = 0

DIM Env AS ChartEnvironment   ' See CHRTB.BI for declaration of
                              ' the ChartEnvironment type.
DIM MonthCategories(1 TO MONTHS) AS STRING ' Array for categories
DIM OJvalues(1 TO MONTHS) AS SINGLE        ' Array for 1st data series
DIM Exploded(1 TO MONTHS) AS INTEGER' "Explode" flags array
                                   ' (specifies which pie slices
                                   ' are separated).
' Initialize the data arrays.
FOR index = 1 TO MONTHS: READ OJvalues(index): NEXT index
FOR index = 1 TO MONTHS: READ MonthCategories$(index): NEXT index

' Set elements of the array that determine separation of pie slices
FOR Flags = 1 TO MONTHS                        ' If value of OJvalues(Flags)
   Exploded(Flags) = (OJvalues(Flags) >= 100) ' >= 100 the corre-
                                              ' sponding flag is set
                                              ' true, separating slices.
NEXT Flags

' Pass the value returned by the BestMode function to the Presentation
' Graphics routine ChartScreen to set the graphics mode for charting.
```

```
ChartScreen (BestMode) ' Even if SCREEN is already set to an acceptable
                       ' mode, you still must set it with ChartScreen.

' Check to make sure ChartScreen succeeded:
IF ChartErr = cBadScreen THEN
   PRINT "Sorry, there is a screen-mode problem in the chart library"
   END
END IF

' Initialize a default pie chart. Pass Env (the environment variable),
DefaultChart Env, cPie, cPercent     ' the constant cPie (for Pie Chart)
                                     ' and cPercent (label slices with
                                     ' percentage).

' Add Titles and some chart options. These assignments modify some
' default values set in the variable Env (of type ChartEnvironment)
' by DefaultChart.

Env.MainTitle.Title = "Good Neighbor Grocery"  ' Specifies the title,
Env.MainTitle.TitleColor = 15                   ' color of title text,
Env.MainTitle.Justify = cCenter              ' alignment of title text,
Env.SubTitle.Title = "Orange Juice Sales"    ' text of chart subtitle,
Env.SubTitle.TitleColor = 11                 ' color of subtitle text,
Env.SubTitle.Justify = cCenter               ' alignment of subtitle text,
Env.ChartWindow.Border = cYes                ' and presence of a border.

' Call the pie-charting routine --- Arguments for call to ChartPie are:
'  Env            Environment variable
'  MonthCategories()Array containing Category labels
'  OJvalues()     Array containing Data values to chart
'  Exploded()     Integer array tells which pieces of the pie should
'                 be separated (non-zero=exploded, 0=not exploded)
'  MONTHS         Tells number of data values to chart
   ChartPie Env, MonthCategories(), OJvalues(), Exploded(), MONTHS
   SLEEP
' If the rest of your program isn't graphic, you could reset original
' video mode here.
END

' Simulate data generation for chart values and category labels.
DATA 33,27,42,64,106,157,182,217,128,62,43,36
DATA "Jan","Feb","Mar","Apr","May","Jun","Jly","Aug","Sep","Oct","Nov"
DATA "Dec"
```

```
'========== Function to determine and set highest resolution ========
' The BestMode function uses a local error trap to check available
' modes, then assigns the integer representing the best mode for
' charting to its name so it is returned to the caller. The function
' terminates execution if the hardware doesn't support a mode
' appropriate for Presentation Graphics.
'=====================================================================
FUNCTION BestMode

' Set a trap for expected local errors -- handled within the function.
ON LOCAL ERROR GOTO ScreenError

FOR TestValue = HIGHESTMODE TO 0 STEP -1
   DisplayError = FALSE
   SCREEN TestValue
   IF DisplayError = FALSE THEN
      SELECT CASE TestValue
         CASE 0
            PRINT "Sorry, you need graphics to display charts"
            END
         CASE 12, 13
            BestMode = 12
         CASE 2, 7
            BestMode = 2
         CASE ELSE
            BestMode = TestValue
      END SELECT
      EXIT FUNCTION
   END IF
NEXT TestValue
' Note there is no need to turn off the local error handler.
' It is turned off automatically when control passes out of
' the function.

EXIT FUNCTION
```

```
'==================  Local error handler code ====================
' The ScreenError label identifies a local error handler referred to
' above. Invalid SCREEN values generate Error # 5 (Illegal
' function call) --- so if that is not the error reset ERROR to the
' ERR value that was generated so the error can be passed to other,
' possibly more appropriate, error-handling routine.
' ================================================================
ScreenError:
   IF ERR = 5 THEN
      DisplayError = TRUE
      RESUME NEXT
   ELSE
      ERROR ERR
   END IF
END FUNCTION
```

Output

The pie chart in Figure 6.3 remains on the screen until a key is pressed.

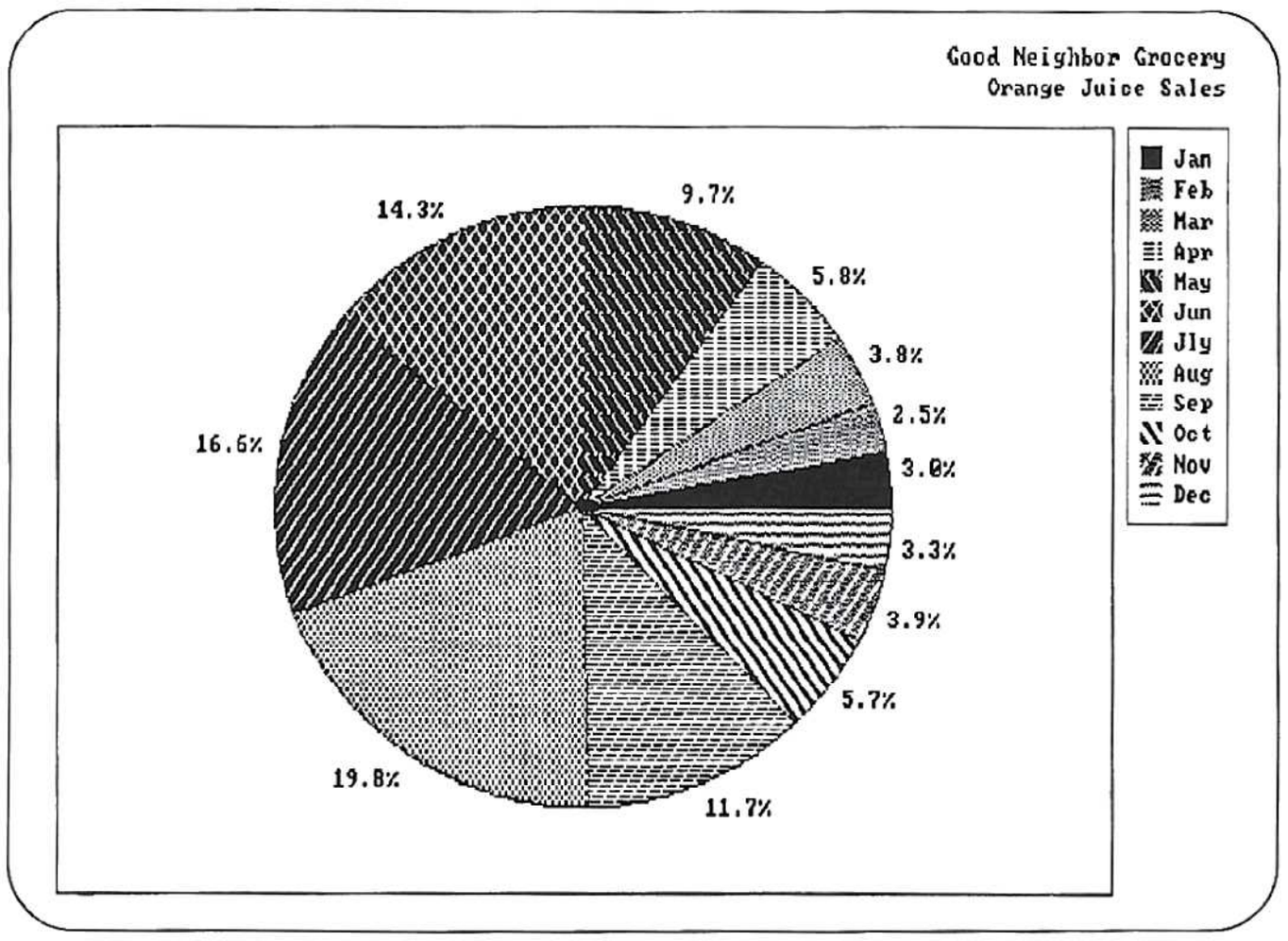


Figure 6.3 Pie Chart

Bar Chart

The code for PGPIE.BAS needs only the following alterations to produce bar, column, and line charts for the same data:

- Give new arguments to **DefaultChart** that specify chart type and style.
- Assign Titles for the x-axis and y-axis in the `Env` structure (this is optional).
- Replace the call to **ChartPie** with **Chart**. This function produces bar, column, and line charts depending on the value of the second argument to **DefaultChart**.
- Remove references to the Exploded flag array (applicable only to pie charts).

Example PGBAR.BAS is the module-level code for a program to display a bar chart. It calls the same procedure (`BestMode`) listed with the preceding PGPIE.BAS example. The following example produces the bar chart shown in Figure 6.4.

```
' PGBAR.BAS: Creates sample bar chart

DEFINT A-Z
' $INCLUDE: 'CHRTB.BI'
DECLARE FUNCTION BestMode ()
CONST FALSE = 0, TRUE = NOT FALSE, MONTHS = 12
CONST HIGHESTMODE = 13, TEXTONLY = 0

DIM Env AS ChartEnvironment   ' See CHRTB.BI for declaration of
                              ' the ChartEnvironment type
DIM MonthCategories(1 TO MONTHS) AS STRING  ' Array for categories
                                            ' (used for pie, column,
                                            ' and bar charts).

DIM OJvalues(1 TO MONTHS) AS SINGLE    ' Array for data series.

' Initialize the data arrays
FOR index = 1 TO MONTHS: READ OJvalues(index): NEXT index
FOR index = 1 TO MONTHS: READ MonthCategories$(index): NEXT index

' Pass the value returned by the BestMode function to the Presentation
' Graphics routine ChartScreen to set the graphics mode for charting.
```

```
ChartScreen (BestMode) ' Even if SCREEN is already set to an acceptable
                       ' mode, you still must set it with ChartScreen.
                       ' Check to make sure ChartScreen succeeded.IF
ChartErr = cBadScreen THEN
   PRINT "Sorry, there is a screen-mode problem in the chart library."
   END
END IF
' Initialize a default pie chart
DefaultChart Env, cBar, cPlain  ' Pass Env (the environment variable),
                                ' the constant cBar (for Bar Chart)
                                ' and cPlain.

' Add Titles and some chart options. These assignments modify some
' default values set in the variable Env (of type ChartEnvironment)
' by DefaultChart.

Env.MainTitle.Title = "Good Neighbor Grocery" ' Specifies text of
                                  ' the chart title,
Env.MainTitle.TitleColor = 15     ' color of title text,
Env.MainTitle.Justify = cRight    ' alignment of title text,
Env.SubTitle.Title = "Orange Juice Sales"  ' text of chart subtitle,
Env.SubTitle.TitleColor = 15              ' color of subtitle text,
Env.SubTitle.Justify = cRight     ' alignment of subtitle text,
Env.ChartWindow.Border = cNo      ' and absence of a border.

' The next 2 assignments label the x-axis and y-axis
Env.XAxis.AxisTitle.Title = "Quantity (cases)"
Env.YAxis.AxisTitle.Title = "Months"

' Call the bar-charting routine --- Arguments for call to Chart are:
' Env                 Environment variable
' MonthCategories()   Array containing Category labels
' OJvalues()          Array containing Data values to chart
' MONTHS              Tells number of data values to chart

   Chart Env, MonthCategories(), OJvalues(), MONTHS
   SLEEP
   ' If the rest of your program isn't graphic,
   ' reset original screen mode here.
END

' Simulate data generation for chart values and category labels
DATA 33,27,42,64,106,157,182,217,128,62,43,36
DATA "Jan","Feb","Mar","Apr","May","Jun","Jly","Aug","Sep","Oct"
DATA "Nov","Dec"
```

Output

The PGBAR.BAS example produces the chart shown in Figure 6.4.

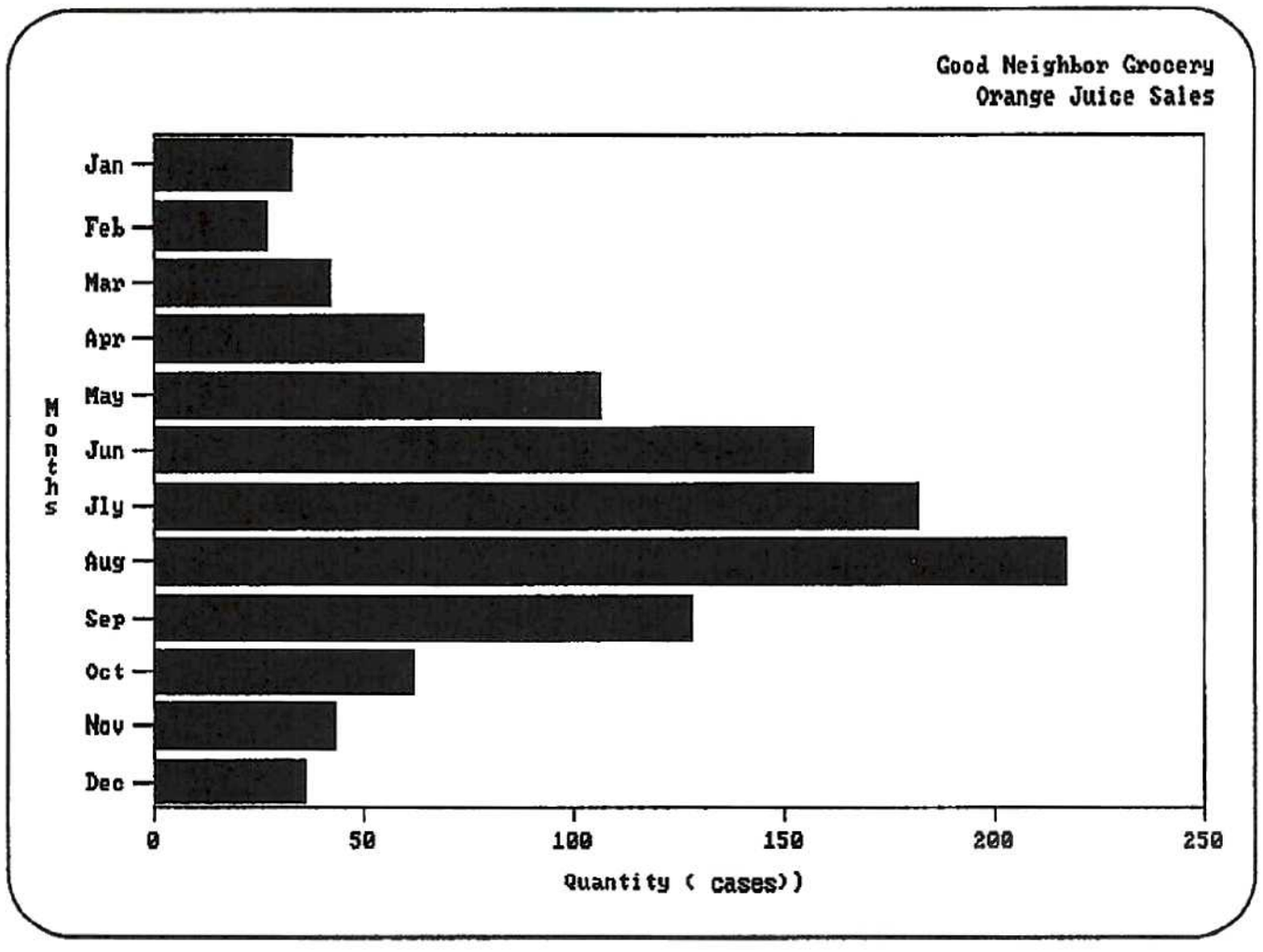


Figure 6.4 Bar Chart

Line and Column Charts

You could turn the grocer's bar chart into a line chart in two easy steps. Simply specify the new chart type when calling **DefaultChart** and switch the axis `Titles`. To produce a line chart for the data, replace the call to **DefaultChart** with:

```
DefaultChart(Env, cLine, cLines)
```

The constant `cLine` specifies a line chart, and the constant `cLines` specifies that the points are to be joined by lines. If you pass `cNoLines` as the third argument, the points would appear, but would not be connected. To switch the labels on the axes, just replace the assignments to the axis labels as follows:

```
Env.XAxis.AxisTitle.Title="Months"
Env.YAxis.AxisTitle.Title="Quantity (cases)"
```

Output

Notice that now the x-axis is labelled "Months" and the y-axis is labelled "Quantity (cases)." Figure 6.5 shows the resulting line chart.

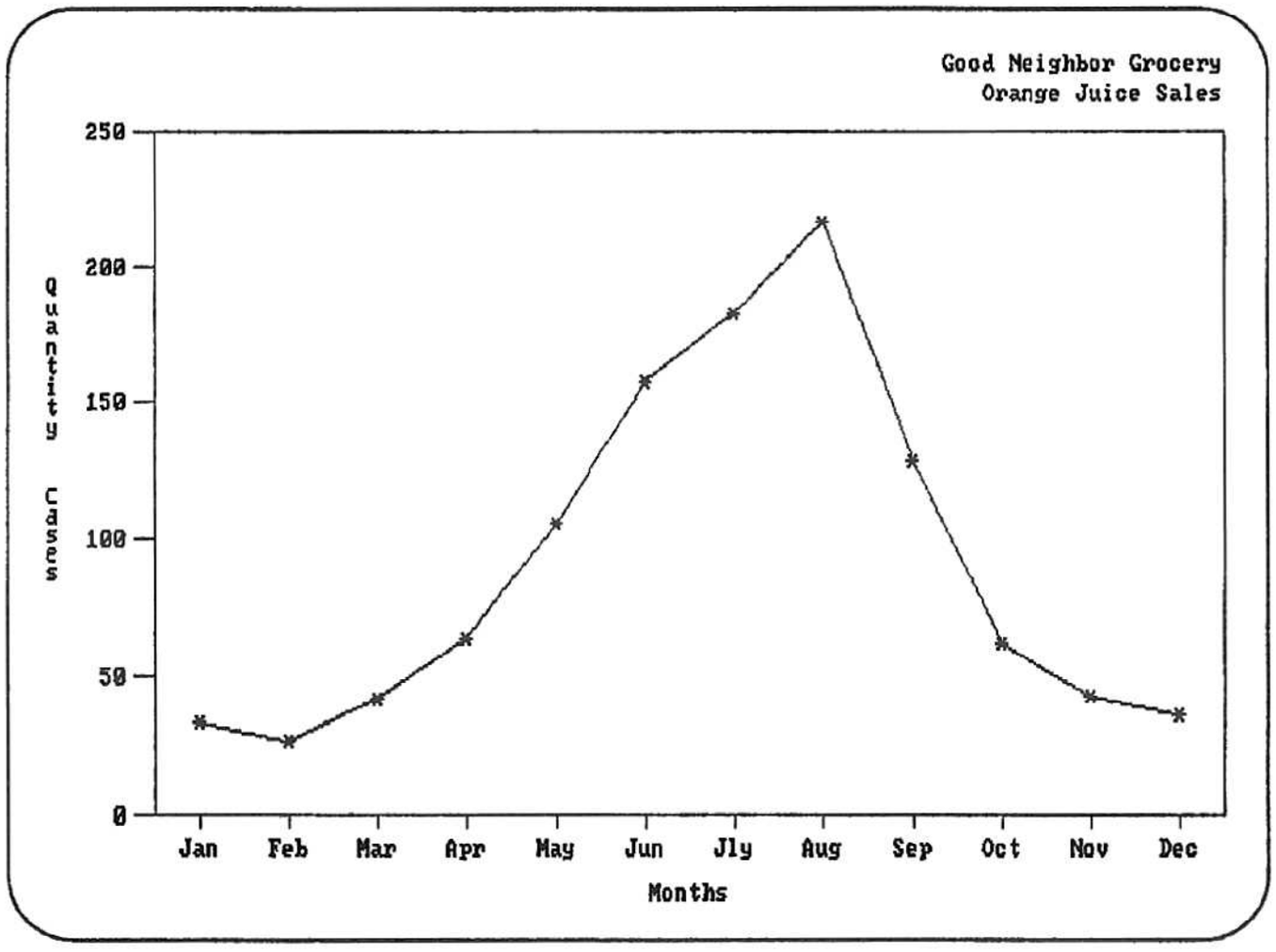


Figure 6.5 Line Chart

Creating an equivalent column chart requires only one change. Use the same code as for the line chart and replace the call to **DefaultChart** with:

```
DefaultChart( Env, cColumn, cPlain )
```

Output

Figure 6.6 shows the column chart for the grocer's data.

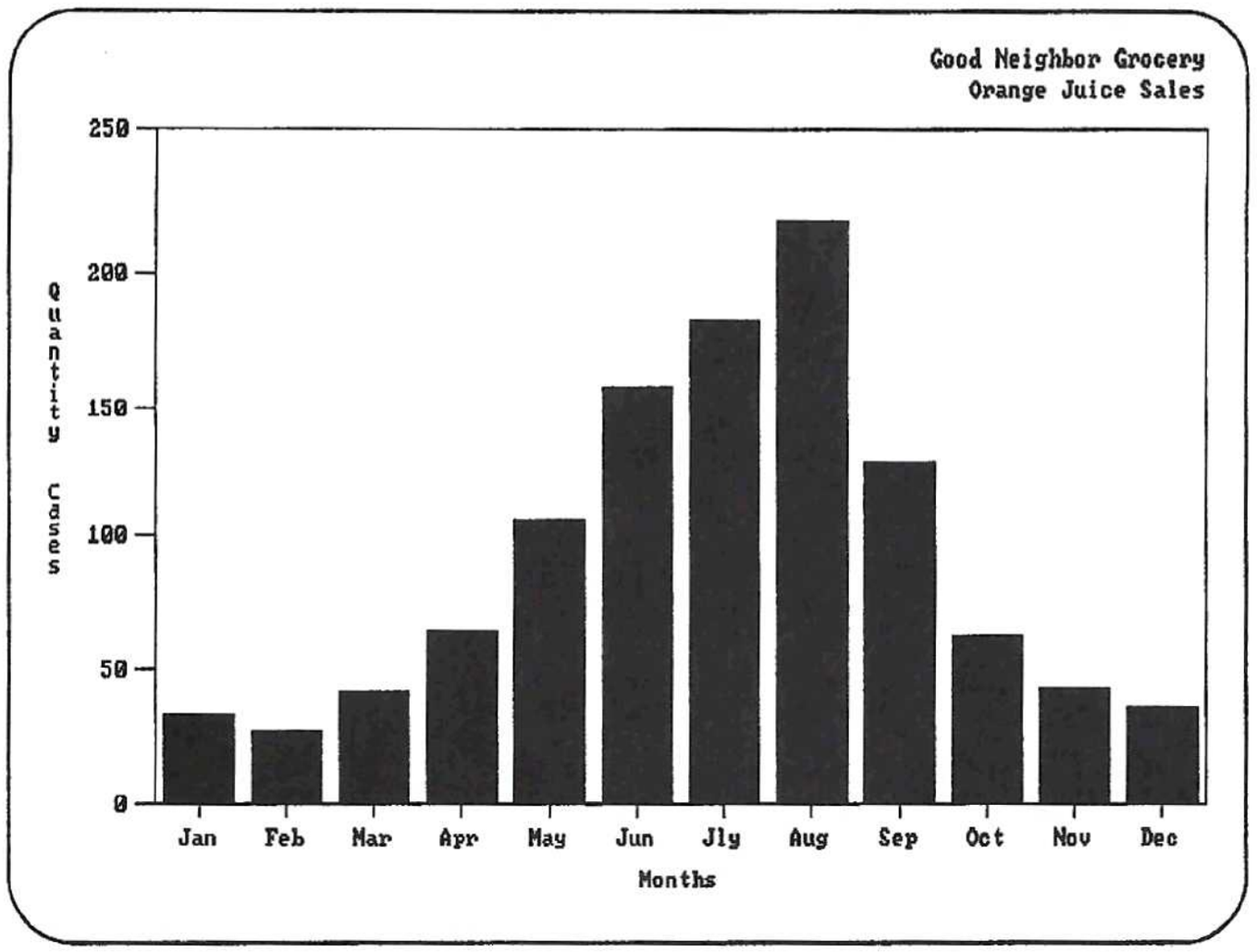


Figure 6.6 Column Chart

Scatter Diagram

Now suppose that the store owner wants to compare the sales of orange juice to the sales of another product, say hot chocolate. Table 6.4 shows a tabular comparison.

Table 6.4 Good Neighbor Grocery Orange Juice and Hot Chocolate Sales

Months	Orange Juice (cases)	Hot Chocolate (cases)
January	33	37
February	27	27
March	42	30
April	64	19
May	106	10
June	157	5
July	182	2
August	217	1
September	128	7

Table 6.4 *Continued*

Months	Orange Juice (cases)	Hot Chocolate (cases)
October	62	15
November	43	28
December	36	39

Example PGSCAT.BAS is the module-level code for a program to display a scatter diagram that illustrates the relationship between the sales of orange juice and hot chocolate throughout a 12-month period. It calls the same function (`BestMode`) listed with the PIE.BAS example preceding. Note that the data array `HCvalues` replaces the `MonthCategories` array in this example.

```
' PGSCAT.BAS: Create sample scatter diagram.

DEFINT A-Z
' $INCLUDE: 'CHRTB.BI'
DECLARE FUNCTION BestMode ()
CONST FALSE = 0, TRUE = NOT FALSE, MONTHS = 12
CONST HIGHESTMODE = 13, TEXTONLY = 0

DIM Env AS ChartEnvironment   ' See CHRTB.BI for declaration of the
                              ' ChartEnvironment type
DIM OJvalues(1 TO MONTHS) AS SINGLE ' Array for 1st data series
DIM HCvalues(1 TO MONTHS) AS SINGLE ' Array for 2nd data series

' Initialize the data arrays
FOR index = 1 TO MONTHS: READ OJvalues(index): NEXT index
FOR index = 1 TO MONTHS: READ HCvalues(index): NEXT index

' Pass the value returned by the BestMode function to the Presentation
' Graphics routine ChartScreen to set the graphics mode for charting.

ChartScreen (BestMode) ' Even if SCREEN is already set to an
                       ' acceptable mode, you still have to
                       ' set it with ChartScreen.
IF ChartErr = cBadScreen THEN ' Make sure ChartScreen succeeded.
   PRINT "Sorry, there is a screen-mode problem in the chart library"
   END
END IF
```

```
                                        ' Initialize a default pie chart.
                                        ' Pass Env (the environment
DefaultChart Env, cScatter, cNoLines    ' variable), constant cScatter
                                        ' (for scatter chart),
                                        ' cNoLines (unjoined points).

' Add Titles and some chart options. These assignments modify some
' default values set in the variable Env (of type ChartEnvironment)
' by DefaultChart.

Env.MainTitle.Title = "Good Neighbor Grocery" ' Specify chart title,
Env.MainTitle.TitleColor = 11                 ' color of title text,
Env.MainTitle.Justify = cRight                ' alignment of title text,
Env.SubTitle.Title = "OJ vs. Hot Chocolate"   ' text of chart subtitle,
Env.SubTitle.TitleColor = 15                  ' color of subtitle text,
Env.SubTitle.Justify = cRight                 ' alignment of subtitle text,
Env.ChartWindow.Border = cNo                  ' and absence of a border.

' The next two assignments label the x and y axes of the chart
Env.XAxis.AxisTitle.Title = "Orange Juice Sales"
Env.YAxis.AxisTitle.Title = "Hot Chocolate Sales"

' Call the pie-charting routine --- Arguments for call to ChartPie are:
'  Env    Environment variable
'  OJvalues  Array containing orange-juice sales values to chart
'  HCvalues  Array containing hot-chocolate sales values to chart
'  MONTHS    Number of data values to chart

   ChartScatter Env, OJvalues(), HCvalues(), MONTHS
   SLEEP
' If the rest of your program isn't graphic, you could
' reset original screen mode here.
END

' Simulate data generation for chart values and category labels.
DATA 33,27,42,64,106,157,182,217,128,62,43,36
DATA 37,37,30,19,10,5,2,1,7,15,28,39
```

Output

Figure 6.7 shows the results of PGSCAT.BAS. Notice that the scatter points form a slightly curved line, indicating a correlation exists between the sales of the two products. The store owner can conclude from the scatter diagram that the demand for orange juice is roughly the inverse of the demand for hot chocolate.

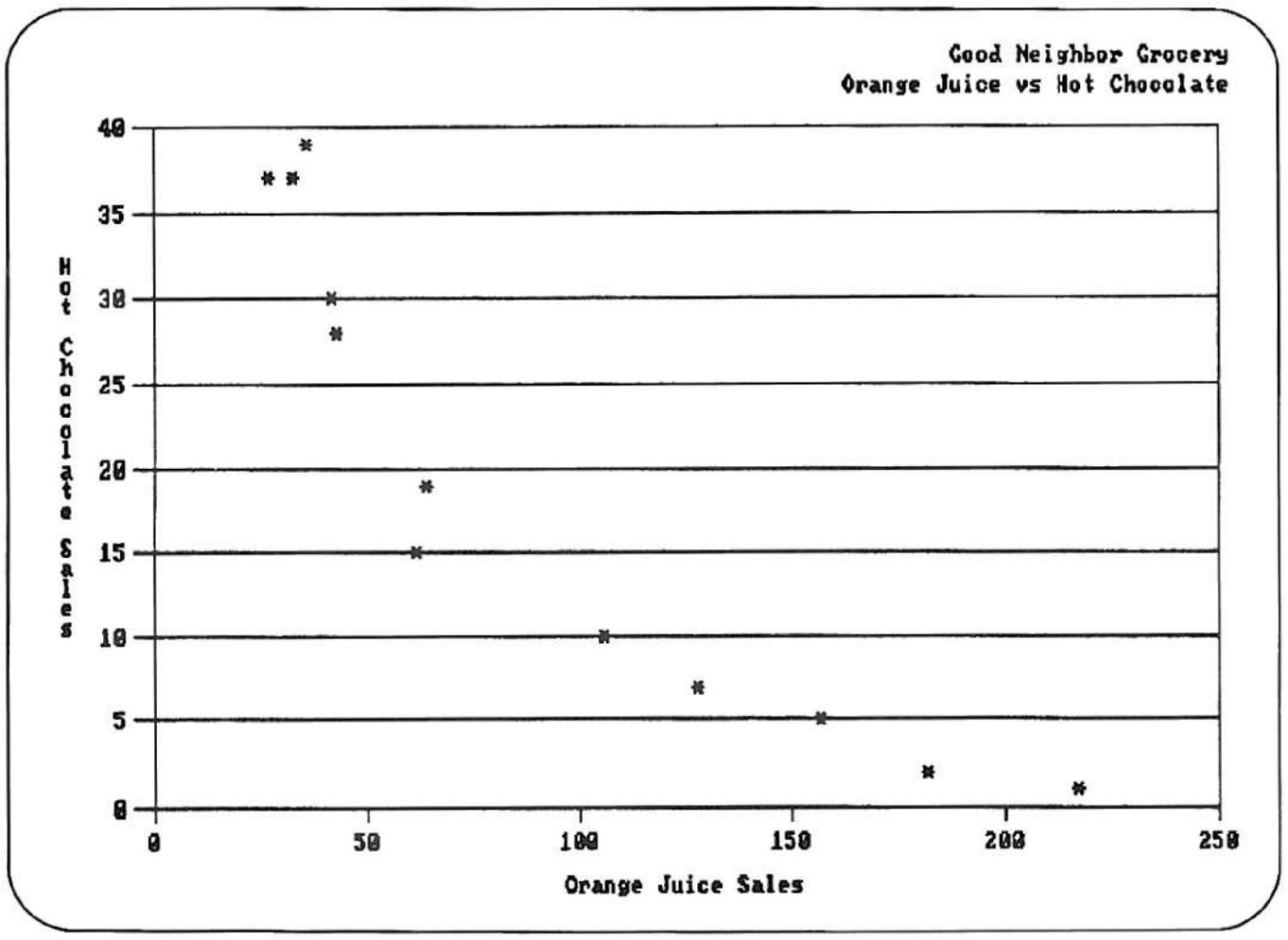


Figure 6.7 Chart Produced by PGSCAT.BAS

Customizing Presentation Graphics

The Presentation Graphics toolbox is built for flexibility. In the preceding examples, you saw how easy it was to change subtitles and axis labels with simple assignment statements. Other elements of the environment can be modified and customized just as easily. You can use its system of default values to produce professional-looking charts with a minimum of programming effort. Or, you can fine-tune the appearance of your charts by overriding default values and initializing variables explicitly in your program. The following section describes all the user-defined data types in the Presentation Graphics toolbox so you can decide which characteristics to accept as supplied and which ones you want to modify. Modification involves declaring a variable of the specified type, then assigning values to its constituent elements. These modifications are always done between the call to **DefaultChart** and the call to the routine that actually displays the chart.

Chart Environment

The include file CHRTB.BI declares a user-defined type, ChartEnvironment, that declares the constituent elements of a structured variable called the "chart environment" variable (`Env` in the preceding examples). The chart environment describes everything about a chart except the actual data to be plotted. The environment determines the appearance of text, axes, grid lines, and legends.

Calling **DefaultChart** fills the chart environment with default values. Presentation Graphics allows you to modify any variable in the environment before displaying a chart. Most initialization of internal Chartlib variables is done through the structured variable you define as having the ChartEnvironment data type, when it is passed to **DefaultChart.**

The sample chart programs provided earlier illustrate how to adjust variables in the chart environment. These programs define a structured variable, `Env`, having the ChartEnvironment data type . The elements of the `Env` structure are the chart environment variables, initialized by the call to **DefaultChart**. Environment variables such as the chart title are then given specific values, as in:

```
Env.MainTitle.Title = "Good Neighbor Grocery"
```

Environment variables that determine colors and line styles deserve special mention. The chart environment holds several such variables which can be recognized by their names. For example, the variable `TitleColor` specifies the color of title text. Similarly, the variable `GridStyle` specifies the line style used to draw the chart grid.

These variables are index numbers, but do not refer directly to the colors or line styles. They correspond instead to palette-entry numbers (Palettes are described later in the chapter). If you set `TitleColor` to 2, Presentation Graphics toolbox uses the color code in the second palette entry to determine the title's color. Thus the title in this case would be the same color as the chart's second data series. If you change the color code in the palette, you'll also change the title's color. You don't have to understand palettes to use the Presentation Graphics toolbox, but understanding how they work gives you greater flexibility in specifying the appearance of charts.

The user-defined type ChartEnvironment has 10 elements, 7 of which are themselves user-defined types. ChartEnvironment is described in detail in the section "ChartEnvironment" later in this chapter.

The next several sections lead up to that description by describing the user-defined types nested within the ChartEnvironment data type. The declaration of ChartEnvironment appearing in CHRTB.BI is as follows:

```
TYPE ChartEnvironment
   ChartType         AS INTEGER
   ChartStyle        AS INTEGER
   DataFont          AS INTEGER
   ChartWindow       AS RegionType
   DataWindow        AS RegionType
   MainTitle         AS TitleType
   SubTitle          AS TitleType
   XAxis             AS AxisType
   YAxis             AS AxisType
   Legend            AS LegendType
END TYPE
```

The remainder of this section describes the chart environment data structure of the Presentation Graphics toolbox. It first examines structures of the four secondary types which make up the chart environment structure. The section concludes with a description of the ChartEnvironment structure type. Each discussion begins with a brief explanation of the structure's purpose, followed by a listing of the structure type definition as it appears in the CHRTB.BI file. All symbolic constants are defined in the file CHRTB.BI .

RegionType

Structures of the type RegionType contain sizes, locations, and color codes for the three windows produced by the Presentation Graphics toolbox: the chart window, the data window, and the legend. Refer to the "Terminology" section earlier in this chapter for definitions of these terms. Placement of the chart window is relative to the screen's logical origin. Placement of the data and legend windows is relative to the chart window.

The CHRTB.BI file defines RegionType as:

```
TYPE RegionType
   X1             AS INTEGER
   Y1             AS INTEGER
   X2             AS INTEGER
   Y2             AS INTEGER
   Background     AS INTEGER
   Border         AS INTEGER
   BorderStyle    AS INTEGER
   BorderColor    AS INTEGER
END TYPE
```

The following table describes the RegionType elements:

Element	Description
`X1, Y1, X2, Y2`	Window (region) coordinates in pixels. The ordered pair (`X1, Y1`) specifies the coordinate of the upper left corner of the window. The ordered pair (`X2, Y2`) specifies the coordinate of the lower right corner. The reference point for the coordinates depends on the type of window. The chart window is located relative to the upper left corner of the screen. The data and legend windows are located relative to the upper left corner of the chart window. This allows you to change the position of the chart window without having to redefine coordinates for the other two windows.
`Background`	An integer between 0 and `cPalLen` representing a palette index (described in the section "Palettes") that specifies the window's background color. The default value for `Background` is 0.
`Border`	A `cYes, cNo` (true/false) variable that determines whether a border frame is drawn around a window.
`BorderStyle`	An integer between 0 and `cPalLen` representing a palette index (described in the section "Palettes") that specifies the line style of the window's border frame. The default value is 1.
`BorderColor`	An integer between 0 and `cPalLen` representing a palette index (described in the section "Palettes") that specifies the color of the window's border frame. The default value is 1.

TitleType

Structures of the type TitleType determine text, color, font, and alignment of titles appearing in the graph. The CHRTB.BI file defines the structure type as:

```
TYPE TitleType
   Title          As STRING * 70
   TitleFont      AS INTEGER
   TitleColor     AS INTEGER
   Justify        AS INTEGER
END TYPE
```

The following list describes TitleType elements:

Element	Description
`Title`	A string containing title text. For example, if `Env` is a structured variable of type ChartEnvironment, then the variable `Env.MainTitle.Title` holds the character string used for the main title of the chart. Similarly, `Env.XAxis.AxisTitle.Title` contains the x-axis title.
`TitleFont`	An integer between 1 and the number of fonts loaded that specifies a title's font. The default value for `TitleFont` is 1.
`TitleColor`	An integer between 0 and `cPalLen` that specifies a title's color. The default value for `TitleColor` is 1.
`Justify`	An integer specifying how the title is placed on its line within the chart window. The symbolic constants defined in the CHRTB.BI file for this variable are `cLeft`, `cCenter`, and `cRight`, meaning flush left, centered, and flush right.

AxisType

Structures of type AxisType contain variables for the axes such as color, scale, grid style, and tick marks. The CHRTB.BI file defines the AxisType structure as:

```
TYPE AxisType
   Grid          AS INTEGER
   GridStyle     AS INTEGER
   AxisTitle     AS TitleType
   AxisColor     AS INTEGER
   Labeled       AS INTEGER
   RangeType     AS INTEGER
   LogBase       AS SINGLE
   AutoScale     AS INTEGER
   ScaleMin      AS SINGLE
   ScaleMax      AS SINGLE
   ScaleFactor   AS SINGLE
   ScaleTitle    AS TitleType
   TicFont       AS INTEGER
   TicInterval   AS SINGLE
   TicFormat     AS INTEGER
   TicDecimals   AS INTEGER
END TYPE
```

The following list describes the elements of the AxisType structure:

Element	Description
`Grid`	A `cYes`, `cNo` (true/false) value that determines whether grid lines are drawn for the associated axis. Grid lines span the data window perpendicular to the axis. One grid line is drawn for each tick mark on the axis.
`GridStyle`	An integer between 0 and `cPalLen` that specifies the grid's line style. Lines can be solid, dashed, dotted, or some combination. Grid styles are drawn from *PaletteB%*, described in the section "Palettes," later in this chapter. The default value for `GridStyle` is 1. Note that the color of the parallel axis determines the color of the grid lines. Thus the x-axis grid is the same color as the y-axis, and the y-axis grid is the same color as the x-axis.
`AxisTitle`	A TitleType structure that defines the title of the associated axis. The title of the y-axis displays vertically to the left of the y-axis, and the title of the x-axis displays horizontally below the x-axis.

`AxisColor`	An integer between 0 and `cPalLen` that specifies the color used for the axis and parallel grid lines. (See the preceding description for `GridStyle`.) Note that this member does not determine the color of the axis title. That selection is made through the structure `AxisTitle`. The default value is 1.
`Labelled`	A `cYes`, `cNo` (true/false) value that determines whether tick marks and labels are drawn on the axis. Axis labels should not be confused with axis titles. Axis labels are numbers or descriptions such as "23.2" or "January" attached to each tick mark.
`RangeType`	An integer that determines whether the scale of the axis is linear or logarithmic. The `RangeType` variable applies only to value data (not to category labels). Specify a linear scale with the `cLinearAxis` constant. A linear scale is best when the difference between axis minimum and maximum is relatively small. For example, a linear axis range 0–10 results in 10 tick marks evenly spaced along the axis. Use `cLogAxis` to specify a logarithmic `RangeType`. Logarithmic scales are useful when the data varies exponentially. Line graphs of exponentially varying data can be made straight with a logarithmic `RangeType`.
`LogBase`	If `RangeType` is logarithmic, the `LogBase` variable determines the log base used to scale the axis. Default value is 10.
`AutoScale`	A `cYes`, `cNo` (true/false) variable. If `AutoScale` is `cYes`, the Presentation Graphics toolbox automatically determines values for `ScaleMin`, `ScaleMax`, `ScaleFactor`, `ScaleTitle`, `TicInterval`, `TicFormat`, and `TicDecimals` (see the following). If `AutoScale` equals `cNo`, these seven variables must be specified in your program.
`ScaleMin`	Lowest value represented by the axis.
`ScaleMax`	Highest value represented by the axis.

`ScaleFactor`

All numeric data is scaled by dividing each value by `ScaleFactor`. For relatively small values, the variable `ScaleFactor` should be 1, which is the default. But data with large values should be scaled by an appropriate factor. For example, data in the range 2 million–20 million should be plotted with `ScaleMin` set to 2, `ScaleMax` set to 20, and `ScaleFactor` set to 1 million.

If `AutoScale` is set to `cYes`, the Presentation Graphics toolbox automatically determines a suitable value for `ScaleFactor` based on the range of data to be plotted. The Presentation Graphics toolbox selects only values that are a factor of 1000—that is, values such as one thousand, one million, or one billion. It then labels the `ScaleTitle` appropriately (see the following). If you desire some other value for scaling, you must set `AutoScale` to `cNo` and set `ScaleFactor` to the desired scaling value.

`ScaleTitle`

A `TitleType` structure defining the attributes of a title. If `AutoScale` is `cYes`, Presentation Graphics toolbox automatically writes a scale description to `ScaleTitle`. If `AutoScale` equals `cNo` and `ScaleFactor` is 1, `ScaleTitle.Title` should be blank. Otherwise your program should copy an appropriate scale description to `ScaleTitle.Title`, such as "(x 1000)," "(in millions of units)," "times 10 thousand dollars," etc.
For the y-axis, the `Scaletitle` text displays vertically between the axis title and the y-axis. For the x-axis the scale title appears below the x-axis title.

Note: The following four variables apply to axes with value data. `TicFont` also applies to category labels; the remainder are ignored for the category axis.

`TicFont`

An integer between 1 and the total number of fonts loaded specifying which of a group of currently loaded fonts to use for this axis's tick marks. The default value is 1.

`TicInterval`

Sets interval between tick marks on the axis. The tick interval is measured in the same units as the numeric data associated with the axis. For example, if two sequential tick marks correspond to the values 20 and 25, the tick interval between them is 5.

`TicFormat`

An integer that determines the format of the labels assigned to each tick mark. Set `TicFormat` to `cExpFormat` for exponential format or to `cDecFormat` for decimal (the default).

`TicDecimals`

Number of digits to display after the decimal point in tick labels. Maximum value is 9.

LegendType

Structured variables of the LegendType user-defined type contain size, location, and colors of the chart legend. The CHRTB.BI file defines the elements of LegendType as:

```
TYPE LegendType
   Legend        AS INTEGER
   Place         AS INTEGER
   TextColor     AS INTEGER
   TextFont      AS INTEGER
   AutoSize      AS INTEGER
   LegendWindow  AS RegionType
```

The following table describes LegendType elements:

Element	Description
`Legend`	A `cYes`, `cNo` (true/false) variable that determines whether a legend is to appear on a multi-series chart. Pie charts always have a legend. The `Legend` variable is ignored by routines that graph other single-series charts.
`Place`	An integer that specifies the location of the legend relative to the data window. Setting the variable Place equal to the constant `cRight` positions the legend to the right of the data window. Setting `Place` to `cBottom` positions the legend below the data window. Setting `Place` to `cOverLay` positions the legend within the data window. These settings influence the size of the data window. If `Place` equals `cBottom` or `cRight`, Presentation Graphics toolbox automatically sizes the data window to accommodate the legend. If `Place` equals `cOverlay` the data window is sized without regard to the legend.
`TextColor`	An integer between 0 and `cPalLen` that specifies the color of text within the legend window.
`TextFont`	An integer specifying which of a group of currently loaded fonts to use for the legend text.

AutoSize	A cYes, cNo (true/false) variable that determines whether the Presentation Graphics toolbox will automatically calculate the size of the legend. If AutoSize equals cNo, the legend window must be specified in the LegendWindow structure (see the following).
LegendWindow	A RegionType structure that defines coordinates, background color, and border frame for the legend. Coordinates given in LegendWindow are ignored (and overwritten) if AutoSize is cYes.

ChartEnvironment

A structured variable of the ChartEnvironment type defines the chart environment. The following listing shows that a ChartEnvironment type structure consists almost entirely of structures of the four types discussed in the preceding sections.

The CHRTB.BI file defines the ChartEnvironment structure type as:

```
TYPE ChartEnvironment
   ChartType     AS INTEGER
   ChartStyle    AS INTEGER
   DataFont      AS INTEGER
   ChartWindow   AS RegionType
   DataWindow    AS RegionType
   MainTitle     As TitleType
   SubTitle      As TitleType
   XAxis         As AxisType
   YAxis         As AxisType
   Legend        As LegendType
END TYPE
```

The following list describes ChartEnvironment elements:

Element	Description
ChartType	An integer that determines the type of chart displayed. The value of the variable ChartType is either cBarChart, cColumnChart, cLineChart, cScatterChart, or cPieChart. This variable is set from the second argument for the **DefaultChart** routine.

`ChartStyle`	An integer that determines the style of the chart. Legal values for `ChartStyle` are `cPercent` and `cNoPercent` for pie charts; `cStacked` and `cPlain` for bar and column charts; and `cLines` and `cPoints` for line graphs and scatter diagrams. This variable corresponds to the third argument for the **DefaultChart** routine.
`DataFont`	An integer that identifies the font to use in drawing the plotting characters in line charts and scatter charts. The range of possible values depends on how many fonts are loaded. See the section "Loading Graphics Fonts" later in this chapter, for information on loading fonts.
`ChartWindow`	A RegionType structure that defines the appearance of the chart window.
`DataWindow`	A RegionType structure that defines the appearance of the data window.
`MainTitle`	A TitleType structure that defines the appearance of the main title of the chart.
`SubTitle`	A TitleType structure that defines the appearance of the chart's Subtitle.
`XAxis`	An AxisType structure that defines the appearance of the x-axis. (This variable is not applicable for pie charts.)
`YAxis`	An AxisType structure that defines the appearance of the y-axis. (This variable is not applicable for pie charts.)
`Legend`	A LegendType structure that defines the appearance of the legend window. Applies to multi-series and pie charts. Not applicable to single-series charts.

Note that all the data in a ChartEnvironment type structure is initialized by calling the **DefaultChart** routine. If your program does not call **DefaultChart**, it must explicitly define every variable in the chart environment— a tedious and unnecessary procedure. The recommended method for adjusting the appearance of your chart is to initialize variables for the proper chart type by calling the **DefaultChart** routine, and then reassign selected environment variables such as `Titles`.

Palettes

The Presentation Graphics toolbox displays each data series in a way that makes it discernible from other series. It does this by defining separate "palettes" to determine the color, line style, fill pattern, and plot characters for each different data series in a chart. There is also a palette of line styles used to determine the appearance of window borders and grid lines.

The Presentation Graphics toolbox maintains a set of default palettes. What appears in the default palettes depends on the mode specified in the **ChartScreen** routine, which in turn depends on the BASIC screen modes supported by the host system. Figure 6.8 illustrates the default palettes for a screen mode that permits four colors (indexed as 0,1,2,3).

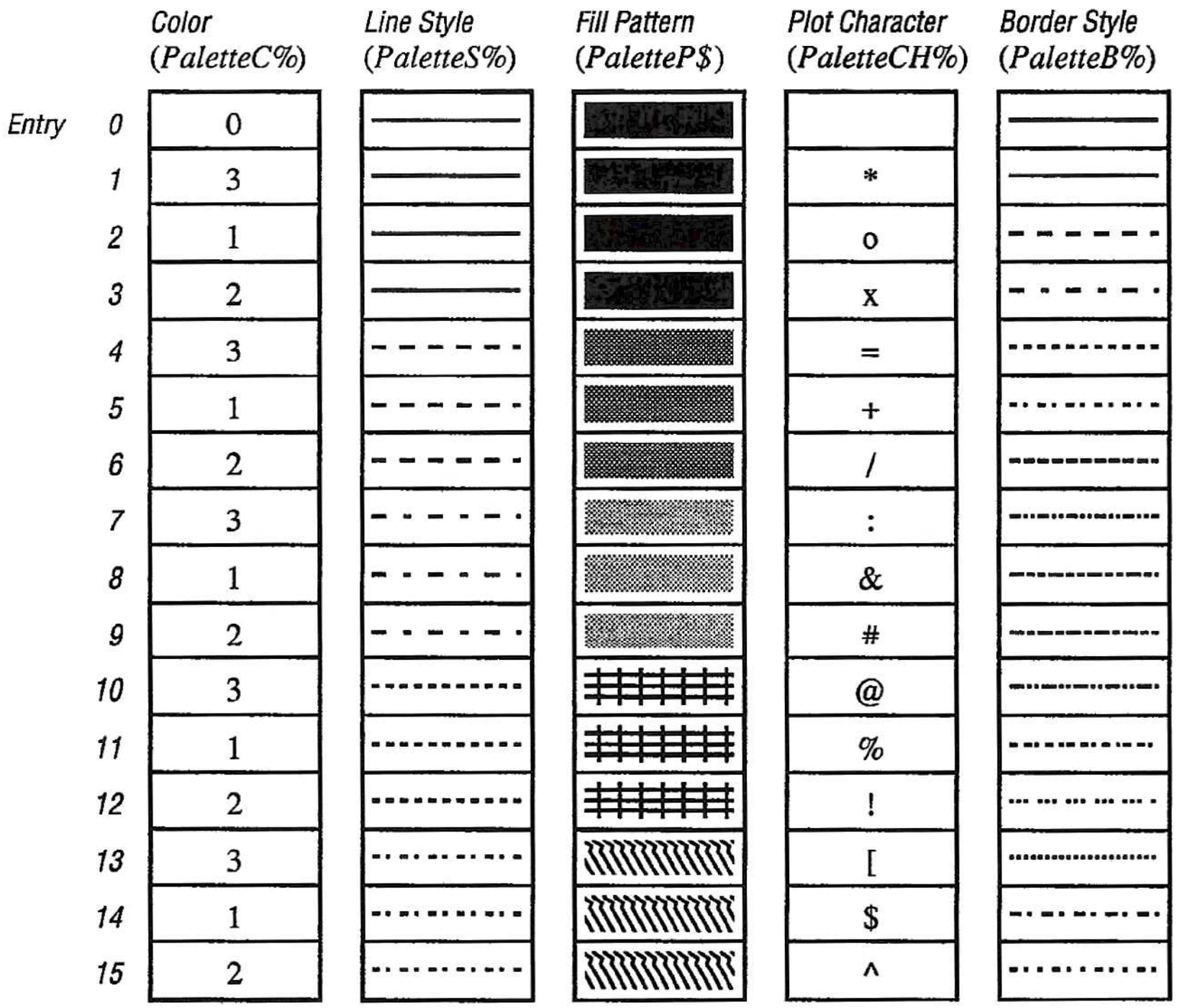


Figure 6.8 Presentation Graphics Toolbox Palettes

Each column in Figure 6.8 represents one of the Presentation Graphics toolbox palettes. When a data series is displayed on a chart, one value from each column in the chart is used to determine the corresponding characteristic. Therefore, each of rows 1–15 in Figure 6.8 represents the characteristics that would be displayed for one of up to 15 data series in a chart displayed on the specified hardware.

Note

Don't confuse the Presentation Graphics toolbox palettes with BASIC's **PALETTE** statement (used to map colors other than the defaults to the color attributes for EGA, VGA, and MCGA adapters). The Presentation Graphics toolbox data series palettes are specific to the Presentation Graphics toolbox.

The Presentation Graphics toolbox provides three routines that you can use to customize the palettes if you wish. Because they are declared in the include file CHRTB.BI (as follows), you can invoke them without a **CALL** statement and without parentheses around the argument list:

GetPaletteDef *PaletteC%( ),PaletteS%( ),PaletteP$( ),PaletteCH%( ),PaletteB%( )*

SetPaletteDef *PaletteC%( ),PaletteS%( ),PaletteP$( ),PaletteCH%( ),PaletteB%( )*

ResetPaletteDef

All the parameters are one-dimensional arrays of length `cPalLen` (starting with the subscript 0 and extending to subscript 15). **GetPaletteDef** lets you access the current palette values. **SetPaletteDef** can be used to substitute custom values within the palette arrays. **ResetPaletteDef** reinstates the default values. **GetPaletteDef** is the **SUB** procedure that gets the array values.

When you invoke **ChartScreen** to reset the video mode, the arrays are initialized to their default values. Once **ChartScreen** has filled the arrays with the default values for the specified screen mode, you can change values in any of the arrays. You use **GetPaletteDef** to transfer the default values to your array variables, then invoke **SetPaletteDef** after you assign your custom values to the arrays you want to change. **ResetPaletteDef** restores the internal chart palette to the original default values, so you need not save the values of your first **GetPaletteDef** call for resetting defaults. If you try to call any of these routines before an initial call to **ChartScreen**, an error is generated. The parameters of **GetPaletteDef** and **SetPaletteDef** are defined in the following table:

Parameter	Definition
PaletteC%( )	Integer array corresponding to color number palette entries. Changes here change colors of items like data lines and text.
PaletteS%( )	Integer array determining appearance of lines in multi-series line graphs (for example, solid, dotted, dashed, etc.).
PaletteP$( )	String array determining the bit-map pattern of the filled-in areas in pie, bar, and column charts.
PaletteCH%( )	Integer array specifying which ASCII character is used on a graph for the plot points of each data series in a multi-series line graphs.
PaletteB%( )	Integer array used to set lines in the display that don't appear as data lines within a graph, for example window borders and grid lines.

The following five sections further describe each of the parameters in the preceding list, and effects of calls to the **SetPaletteDef** routine using non-default values in the arrays.

Note If a pie chart has more than 15 slices, slice 16 will have the same color or fill pattern as slice 2; slice 17 will have the same color or fill pattern as slice 3, and so on. Similarly, if the background is not the default, one slice of the pie may be the same color as the background. See the section "Fill Patterns" later in this chapter for an explanation of how to change these elements.

Colors

One of the elements used to distinguish one data series from another is color. The possible colors correspond to pixel values valid for the current graphics mode. (See column 1 in Figure 6.8. Refer to Chapter 5, "Graphics," for a description of pixel values.) Each row in Figure 6.8 contains pixel values that refer to these available colors. These color codes are the values placed in the *PaletteC%()* array when **ChartScreen** is called. The color code in each entry (i.e. the individual elements of the *PaletteC%()* array) then determines the color used to graph the data series associated with the entry.

Say for example, you have a chart with several lines representing different series of data. The default background color for the chart is represented by the code in the `PaletteC%(0)` element, the color for the first line corresponds to the code in the `PaletteC%(1)` element, the color for the second line corresponds to the code in the `PaletteC%(2)` element, and so forth. These colors are also used for labels, titles, and legends.

The first color is always black, which is the pixel value for the screen background color. The second is always white. The remaining elements are repeating sequences of available pixel values, beginning with 1.

The values in the *PaletteC%()* array are passed to a BASIC color statement by the internal charting routines. Each value represents three things: a display attribute, a pixel value, and a color attribute.

For example, calling `ChartScreen 1` with a CGA system (320 X 200 graphics) provides four colors for display. Pixel values from 0 to 3 determine the possible pixel colors—say, black, cyan, magenta, and white respectively. In this case the first eight available color values would be as follows:

Color index	Pixel value	Color
0	0	Black
1	3	White
2	1	Cyan
3	2	Magenta
4	3	White
5	1	Cyan
6	2	Magenta
7	3	White

Notice that the sequence of available colors repeats from the third element. The first data series in this case would be plotted in white, the second series in cyan, the third series in magenta, the fourth series again in white, and so forth.

Video adapters such as the EGA allow 16 on-screen colors. This allows the Presentation Graphics toolbox to graph more series without duplicating colors.

You can use **SetPaletteDef** to change the color code assignments. For example, in the preceding CGA example, if you didn't want the color red to appear in your chart, you could refill the elements of the *PaletteC%()* array as follows:

```
PaletteC%(3) = 3 : PaletteC%(4) = 1 : PaletteC%(5) = 3 :
PaletteC%(6) = 1 PaletteC%(7) = 3
```

However, if more than two data series appeared in the graph, the lines representing series after the first two would repeat the colors of the first two. To differentiate these lines clearly, you would have to adjust the available line styles (described in the following section "Line Styles") for the palettes. In types of charts other than line charts (e.g. pie charts), changing color assignments would also necessitate modifying the available fill patterns (described in the section "Fill Patterns") as well as the line styles available. Note that the colors in pie, bar, and column charts are not determined by the available colors but by color attributes of the fill patterns. Reassigning the values in the `PaletteC%()` array changes only the outline of a bar, column, or pie slice.

Line Styles

The Presentation Graphics toolbox matches the available colors with a collection of different line styles (column two in Figure 6.8). These are the values in *PaletteS%()* array (the second parameter to the **GetPaletteDef** routine). Entries for the line styles define the appearance of lines such as axes and grids. Lines can be solid, dotted, dashed, or of some combination.

Each palette entry (each entry in the PaletteS% column) is a code that refers to one of the line styles in the same way that each entry in the PaletteC% column is a color code that refers to a color index. The style code value in a palette is applicable only to line graphs and lined scatter diagrams. The style code determines the appearance of the lines drawn between points.

The palette entry's style code adds further variety to the lines of a multi-series graph. It is most useful when the number of lines in a chart exceeds the number of available colors. For example, a graph of nine different data series must repeat colors if only three foreground colors are available for display. However, the style code for each color repetition will be different, ensuring that none of the lines look the same. As mentioned previously, if you modified the repetition pattern for the colors, you would have to adjust the line styles repetition pattern as well. Extending the example of the previous section, if you limited your chart lines to two of the available three foreground colors, you would need to adjust the style pool so that differentiation by type of line would begin earlier. You would do this by changing the *PaletteS%()* array as follows:

```
PaletteS%(0) = &HFFFF : PaletteS%(1) = &HFFFF : PaletteS%(2) = &HFFFF
PaletteS%(3) = &HF0F0 : PaletteS%(4) = &HF0F0 : PaletteS%(5) = &HF060
PaletteS%(6) = &HF060 : PaletteS%(7) = &HCCCC
```

Fill Patterns

The Presentation Graphics toolbox environment also maintains a default collection of fill patterns (column three, *PaletteP$( )*, in Figure 6.8). Fill patterns determine the fill design for column, bar, and pie charts. These are the values in the *PaletteP$( )* array (the third parameter to the **GetPaletteDef** routine). The *PaletteP$( )* array values are used within the Presentation Graphics toolbox as the *paint* parameters to the **PAINT** statement. Manipulating the fill patterns is trickier than the colors and line styles, because the fill patterns determine both the pattern and the color of a fill. The discussion in this section depends on an understanding of information in the section "Painting with Patterns: Tiling" in Chapter 5, "Graphics," which describes how to manipulate "bit maps" in BASIC. The example in the section "Pattern Editor" in that chapter is also instructive.

Each string in the *PaletteP$()* array contains information that is passed as the *paint* parameter to the **PAINT** statement which then defines the fill pattern (and the color of the fill, if color is available), for the data series associated with the palette.

You can change the fill color and pattern for pie, column, and bar charts using the **MakeChartPattern$** and **GetPattern$** routines in combination with the **GetPaletteDef** and **SetPaletteDef** routines.

To change fill pattern and color, first create a one-dimensional string array with 0 to `cPalLen` elements just as with the `PaletteC%()` and `Palettes%()` integer arrays. For example:

```
DIM Fills$(0 to cPalLen)
```

After calling **GetPaletteDef** with `Fills$()` as the third argument, use the **MakeChartPattern$** routine to construct the values to pass as elements of the `Fills$()` array. **MakeChartPattern$** has the following syntax:

**MakeChartPattern$**(*RefPattern$*, *Foreground%*, *Background%*)

Parameter	Description
RefPattern$	A string representing the pixel pattern. The process for creating the string is described in Chapter 5, "Graphics," in the section "Creating a Single-Color Pattern in Screen Mode 2."
Foreground%	Attribute to map to the pixels in *RefPattern$* that are defined as being on.
Background%	Attribute to map to the pixels in *RefPattern$* that are defined as off.

Note that if *Foreground%* and *Background%* are the same value, the fill pattern appears as a solid color.

Constructing a value for *RefPattern$* is difficult. You can simplify the process by using the **GetPattern$** function to change the internally defined pattern. **GetPattern$** has the following syntax:

GetPattern$(*Bits%*, *PatternNum%*)

Parameter	Description
Bits%	Use 2 for screen mode 1, 8 for screen mode 13, and 1 for all other screen modes.
PatternNum%	An integer between 1 and `cPalLen`.

Using these routines, you could reassign the first four fill pattern values as follows:

```
ChartScreen 9
GetPaletteDef Param1%(),Param2%(),Fills$(),Param4%(),Param5%()
Fills$(1) = MakeChartPattern$(GetPattern$(1,8),10,10)
Fills$(2) = MakeChartPattern$(GetPattern$(1,8),11,11)
Fills$(3) = MakeChartPattern$(GetPattern$(1,8),12,12)
Fills$(4) = MakeChartPattern$(GetPattern$(1,8),13,13)
SetPaletteDef Param1%(),Param2%(),Fills$(),Param4%(),Param5%()
```

When you call **SetPaletteDef** and pass `Fills$()` as the third parameter, the first fill pattern is reassigned with the palette 10 fill pattern; the next is reassigned with the palette 11 fill pattern, and so on.

Plot Characters

The plot character parameter (*PaletteCH%( )*) is an array of numbers representing ASCII characters. The array corresponds to the fourth parameter of **GetPaletteDef** (i.e. *PaletteCH%( )*).

Each element of *PaletteCH%( )* represents the character used to plot points on line graphs and scatter diagrams. Each data series uses a different character to distinguish plot points between data series. The plot characters of the palettes are independent of the colors, line styles, and fill patterns. To change which characters are used to plot points of several data series, just pass the new values as the fourth parameter to **SetPaletteDef.** These values are simply the ASCII values of the characters you want to use for each of the lines. For example, the ASCII value for the plus sign is 43, so if you wanted your first plot line to appear as a series of connected plus signs, you could pass 43 as the `PaletteCH%(0)` element.

Border Styles

The border styles parameter is an array of numbers representing the line styles used for some window borders and grid lines. The array corresponds to the fifth parameter of **GetPaletteDef** (i.e. *PaletteB%( )*). As shown in Figure 6.8, each border style is unique. By changing the values in the *PaletteB%()* array you can specify the order in which border styles are assigned. The border-style values are passed by internal charting routines as line style arguments to **LINE** statements. The border styles are independent of all other parts of the palette.

An Overview of the Presentation Graphics Routines

The chapter concludes with a few words about the 19 routines that comprise the Presentation Graphics toolbox. They are listed in Table 6.5 for convenient reference. However, refer to the *BASIC Language Reference* for a description of the routines and their arguments.

Table 6.5 Presentation Graphics Toolbox Routines

Primary routines	Secondary routines	Miscellaneous routines and functions
DefaultChart		LabelchartV
ChartScreen		LabelchartH
Chart	AnalyzeChart	GetPaletteDef
ChartScatter	AnalyzeScatter	SetPaletteDef
ChartPie	AnalyzePie	ResetPaletteDef
ChartMS	AnalyzeChartMS	GetPattern$
ChartScatterMS	AnalyzeScatterMS	MakeChartPattern$

In most cases you need only be concerned with the seven primary routines. These routines initialize variables and display the selected chart types. As demonstrated in example programs earlier in this chapter, you can create very acceptable business charts with programs that call only the first five Presentation Graphics toolbox primary routines.

Multi-Series Plots

The **ChartMS** and **ChartScatterMS** routines deal with displays involving multiple series of data. They are declared in the include file **CHRTB.BI** as follows:

```
ChartMS Env AS ChartEnvironment,Cat$(),Val!(),N%,First%,Last%,_
                                SerLabels$()
ChartScatterMS Env AS ChartEnvironment,Cat$(),ValX!(),ValY!(),N%,_
                                       First%,Last%,Labs$()
```

Parameter	Description
`Env`	Environment variable of ChartEnvironment type.
`Cat$()`	String array of category specifications for x-axis.
`Val!()`	Two-dimensional array of single-precision numbers representing numeric data to be plotted.
`N%`	Number of rows in the two-dimensional array (that is, the number of elements in the first dimension).
`First%`	Integer representing the first column of data (that is, which element of the second array dimension represents the first data series).
`Last%`	Integer representing the final column of data (that is, which element of the second array dimension represents the last data series).
`SerLabels$()` and `Labs$()`	String arrays that describe the labels for each data series in the legend of the chart. There is one string for each of the columns in the `Val!()` array.
`ValX!()`	Two-dimensional array of single-precision numbers representing the x-axis part of each scatter-chart point.
`ValY!()`	Two-dimensional array of single-precision numbers representing the y-axis part of each scatter-chart point.

When you use the **ChartMS** routine, you collect your data into a two-dimensional array. The number of elements in the first dimension is the same as the number of data points in each of the data series (the rows of the two-dimensional array). The number of elements in the second dimension corresponds to the number of data series to be charted (i.e., each data series is a column of the two-dimensional array).

Example The following example displays a multi-series line chart having two data series. Note that the default data-point characters have been replaced using calls to **GetPaletteDef** and **SetPaletteDef**.

```
' PGLINEMS.BAS - Generates a simple multi-series line chart
DEFINT A-Z
'$INCLUDE: 'CHRTB.BI'        ' Declarations and Definitions
DECLARE FUNCTION BestMode% ()

DIM Env AS ChartEnvironment ' Variable to hold environment structure

DIM AxisLabels(1 TO 4) AS STRING       ' Array of categories
DIM LegendLabels(1 TO 2) AS STRING     ' Array of series labels
DIM Values(1 TO 4, 1 TO 2) AS SINGLE  ' Array of values to plot

DIM Col%(0 TO cPalLen)       ' Define arrays to hold values retrieved
DIM Lines%(0 TO cPalLen)     ' with GetPaletteDef. By modifying these
DIM Fill$(0 TO cPalLen)      ' values, then calling SetPaletteDef, you
DIM Char%(0 TO cPalLen)      ' can change colors, plot characters,
DIM Bord%(0 TO cPalLen)      ' borders, line styles, and fill patterns.

' Read the data to display into the arrays
FOR index = 1 TO 2: READ LegendLabels(index): NEXT index
FOR index = 1 TO 4: READ AxisLabels(index): NEXT index

FOR columnindex = 1 TO 2     ' The array has 2 columns, each of
  FOR rowindex = 1 TO 4      ' which has 4 rows. Each column represents
   READ Values(rowindex, columnindex) ' 1 full data series. First,
  NEXT rowindex              ' fill column 1, then fill column 2
NEXT columnindex             ' with values from the last DATA
                             ' statement (below).
CLS
ChartScreen 9                ' Set graphics mode to EGA

' Retrieve current palette settings, then assign some new values
GetPaletteDef Col%(), lines%(), Fill$(), Char%(), Bord%()

 Col%(2) = 15    '  Assign white as color for second-series line.
 Char%(1) = 4    '  Assign CHR$(4) as plot character for 1st line.
 Char%(2) = 18   '  Assign CHR$(18) as plot character for 2nd line.

' Reset the palettes with modified arrays
SetPaletteDef Col%(), Lines%(), Fill$(), Char%(), Bord%()
```

```
' Enter the changes
DefaultChart Env, cLine, cLines        ' Set up multi-series line chart

' Display the chart
ChartMS Env, AxisLabels(), Values(), 4, 1, 2, LegendLabels()

SLEEP          ' Keep it on screen until user presses a key
END

' Simulated data to be shown on chart
DATA "Qtr 1","Qtr 2"
DATA "Admn","Markg","Prodn","Devel"
DATA 38,30,40,32,18,40,20,12
```

The preceding program displays the chart shown in Figure 6.9.

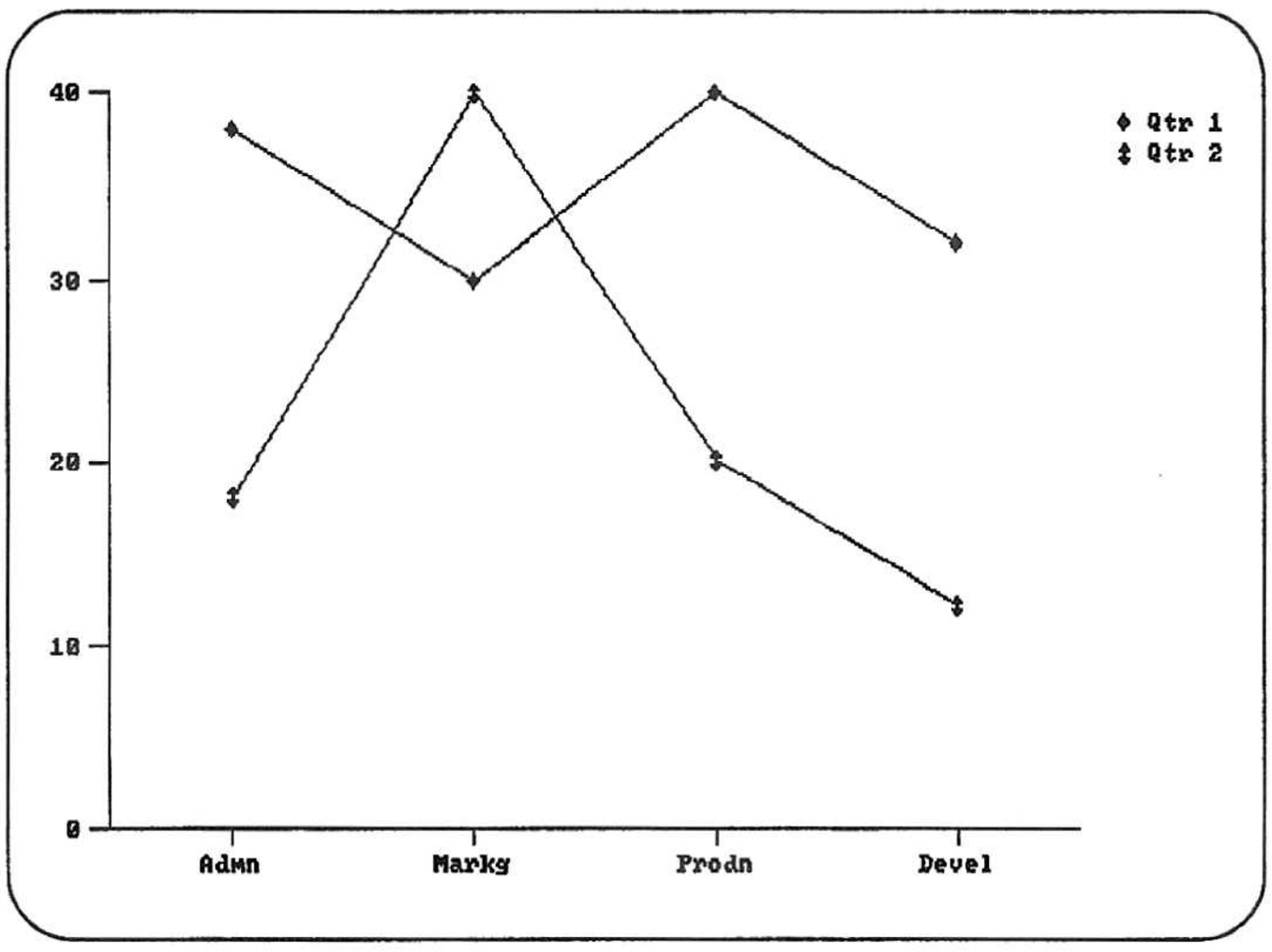


Figure 6.9 Multi-Series Chart

Secondary Routines

The 12 secondary and miscellaneous Presentation Graphics routines do not display charts. Most of them retrieve or set data in the Presentation Graphics chart environment. Two can be used for enhanced labelling of charts.

Analysis Routines

Of special interest among the secondary routines are the analysis routines, identified by the prefix "Analyze" in their names. These five routines calculate default values that pertain to a given chart type and data set. Calling an analysis function has the same effect as calling a corresponding primary function, except that the chart is not displayed. This allows you to pass on to the library the burden of calculating values. You can then make modifications to the resulting values and call a primary routine to display the chart.

Another way to use the analysis routines is to prepare the next chart for fast display while a user is looking at the current chart. For example, you could call the **ChartPie** routine, then place an analysis routine between that call and the **SLEEP** statement that keeps the current display on the screen. The analysis routine would then prepare the next chart for display while the user was looking at the current chart.

After calling an analysis routine, you set the `AutoSize` element of the `Legend` element of your variable of type ChartEnvironment (called `Env` in the previous examples) to `cNo`. Similarly, you set the `AutoScale` elements of the `XAxis` and `YAxis` elements of your variable of type ChartEnvironment to `cNo`, as follows:

```
Env.Legend.AutoSize = cNo
Env.XAxis.AutoScale = cNo
Env.YAxis.AutoScale = cNo
```

When you call the **Chart** routine, the legend and axes are not re-analyzed.

Labelling Routines

Use the **LabelChartH** and **LabelChartV** routines to display text on your chart that is not part of a title or axis label. These routines enable you to attach notes or other messages to your chart. You may also find them useful for labelling separate lines of a multi-series line graph. The next section describes using the font (type styles) toolbox supplied with BASIC. You may find that using special fonts with the **LabelChartH** and **LabelChartV** routines is a handy way to further customize your charts.

Loading Graphics Fonts

When you present your charts, you can specify that the fonts used in various parts of the chart be drawn from a group of fonts that are separately loaded. The routines for doing this (though not the fonts themselves) are included in the CHRTBEFR.QLB that is created by the BASIC Setup program. You can use any group of bit-mapped fonts compatible with Microsoft Windows font libraries and use the routines in FONTB.BAS to register and load them. (Note that vector fonts are not supported.) The FONTB routines let you register, unregister, load, and select fonts that you want to use in your programs, including charting programs. If you don't choose to load custom fonts, the Presentation Graphics toolbox uses its own default font on your charts.

In the multi-series column chart example presented previously, you can use custom fonts by following these steps:

1. Be sure a custom font file, such as TMSRB.FON, is available to your program.
2. Include the file FONTB.BI at the beginning of the main module of the program:

```
'$INCLUDE: 'FONTB.BI'
```

3. Before making any other calls, register and load the custom fonts you want to use:

```
Num% = RegisterFonts("TMSRB.FON")
Num% = LoadFont("N1/N3/N6")
```

4. The functions called in step 3 return integer codes describing how many fonts were registered and loaded. The TMSRB.FON file contains 6 sizes of Times-Roman font (N1,N2,N3,N4,N5,N6), of which only three were specified in `LoadFont`. In the preceding program you could specify any of these fonts for various pieces of text, simply by assigning the numbers 1, 2, or 3 (indicating the N1, N2, or N6 font of the fonts loaded) to the proper elements of the environment variable of ChartEnvironment type. As usual, this needs to be done between the statement that calls **DefaultChart** and the one that actually displays the chart.

```
Env.MainTitle.Title = "Chart in Times-Roman Font"
Env.MainTitle.Font  =  3                          ' Set font size.
Env.Subtitle.Title  = "Looks Nice, Eh!"
Env.Subtitle.Font   =  2                          ' Set font size.
Env.Legend.TextFont =  1                          ' Set font size.
```

The rest of the program is as shown in the preceding example. Once the custom font is registered and loaded, Presentation Graphics toolbox uses that font rather than the default font. See the *BASIC Language Reference* for complete descriptions of each of the FONTB.BAS routines and how to use them.

Chapter 7
Programming with Modules

This chapter shows you gain more control over your programming projects by dividing them into modules. Modules provide a powerful organizing function by letting you divide a program into logically related parts (rather than keeping all the code in one file).

This chapter will show you how to use modules to:

- Write and test new procedures separately from the rest of the program.
- Create libraries of your own **SUB** and **FUNCTION** procedures that can be added to any new program.
- Combine routines from other languages (such as C or MASM) with your BASIC programs.
- Preserve memory when working within QBX.

Why Use Modules?

A module is a file that contains an executable part of your program. A complete program can be contained in a single module, or it can be divided among two or more modules.

In dividing a program into modules, logically related sections are placed in separate files. This organization can speed and simplify the process of writing, testing, and debugging.

Dividing your program into modules has these advantages:

- Procedures can be written separately from the rest of the program, then combined with it. This arrangement is especially useful for testing the procedures, since they can then be checked outside the environment of the program.
- Two or more programmers can work on different parts of the same program without interference. This is especially helpful in managing complex programming projects.
- As you create procedures that meet your own specific programming needs, you can add these procedures to their own module. They can then be reused in new programs simply by loading that module.
- Multiple modules simplify software maintenance. A procedure used by many programs can be in one library module; if changes are needed, the procedure only has to be modified once.
- Modules can contain general-purpose procedures that are useful in a variety of programs (such as procedures that evaluate matrixes, send binary data to a communications port, alter strings, or handle errors). These procedures can be stored in modules, then used in new programs simply by loading the appropriate module into QBX.

Main Modules

The module containing the first executable statement of a program is called the "main module." This statement is never part of a procedure because execution cannot begin within a procedure.

Everything in a module except **SUB** and **FUNCTION** procedures is "module-level code." This includes declarations and definitions, executable program statements and module-level error-handling routines. Figure 7.1 illustrates the relationship between these elements.

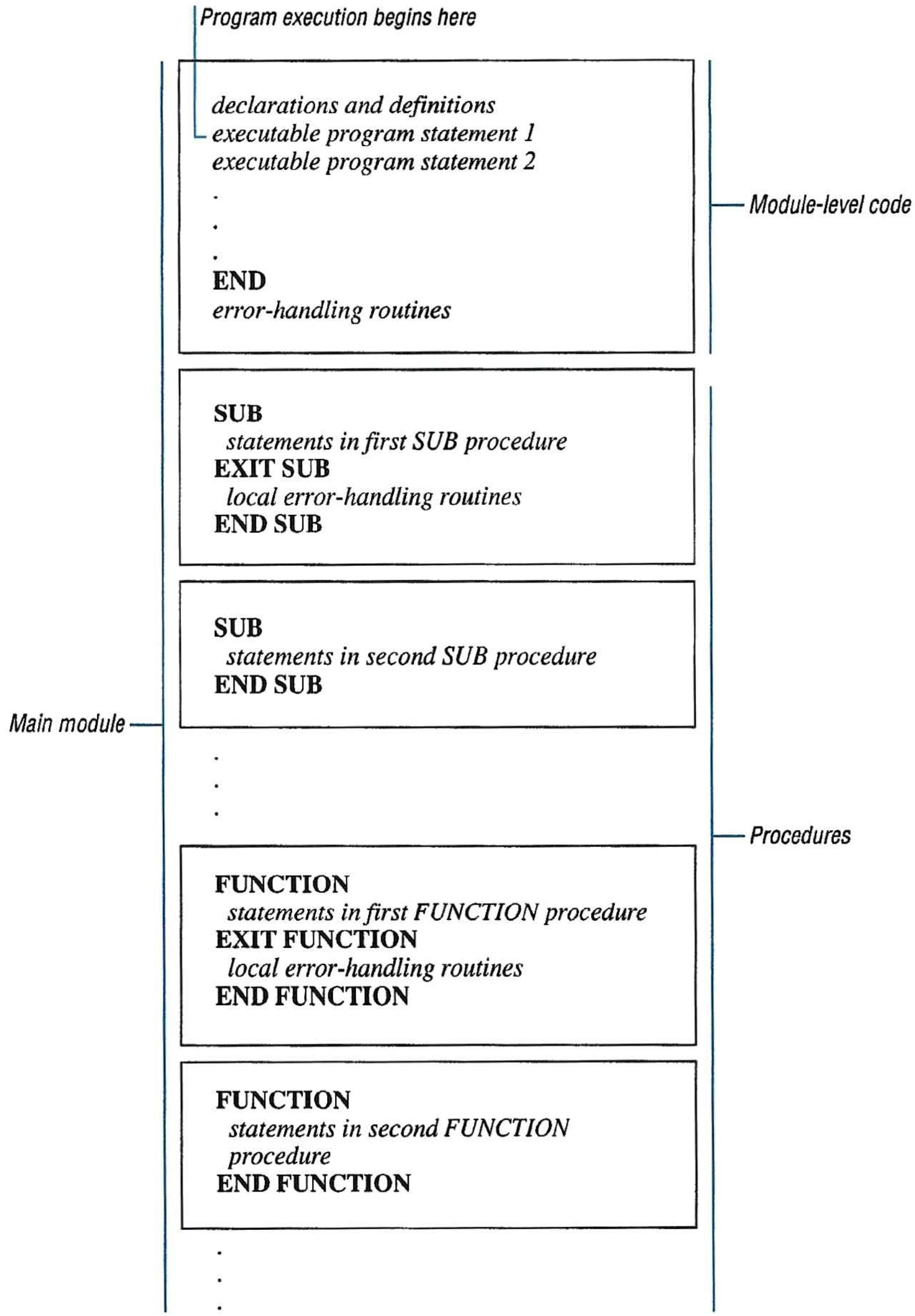


Figure 7.1 Main Module Showing Module-Level Code and Procedures

Non-Main Modules and Procedure-Level Modules

A non-main module can have module-level and procedure-level code. The module-level code consists of metacommands (such as **$INCLUDE**), declarations and definitions (including **COMMON** statements), and any event-trapping or module-level error-handling routine.

A non-main module does not have to contain module-level code. It can consist of nothing but **SUB** and **FUNCTION** procedures. Indeed, such a procedures-only module is the most important use of modules. Here's how to create one:

1. Invoke QBX without opening or loading any files.
2. Write all the **SUB** and **FUNCTION** procedures you wish, but don't enter any module-level code. (Any error- or event-trapping routines and BASIC declarations needed are exceptions.)
3. Choose Save As from the File menu to name and save this module.

To move procedures from one module to another:

1. From QBX, load the files containing the procedures you want to move.
2. If the destination file already exists, choose Load File from the File menu to load it, too. If it doesn't exist, choose Create File from the File menu to make the new file.
3. Choose SUBs from the View menu and the Move option to transfer the procedures from the old to the new file. This transfer is made final when you quit QBX and respond Yes to the dialog box that asks whether you want to save the modified files; otherwise, the procedures remain where they were when you started.

Loading Modules

In QBX you can load as many modules as you wish (limited only by the available memory) by choosing Load File from the File menu. All the procedures in all the loaded modules can be called from any other procedure or from module-level code. No error occurs if a module happens to contain a procedure that is never called.

QBX begins execution with the module-level code of the first module loaded. If you want execution to begin in a different module, choose Set Main Module from the Run menu. Execution halts at the **END** statement in the designated main module.

The ability to choose which module-level code gets executed is useful when comparing two versions of the same program. For example, you might want to test different user interfaces by putting each in a separate module. You can also place test code in a module containing only procedures, then use the Set Main Module command to switch between the program and the tests.

You do not have to keep track of which modules your program uses. Whenever you choose Save All from the File menu, QBX creates (or updates) a .MAK file, which lists all the modules currently loaded. The next time the main module is loaded by choosing Open Program from the File menu, QBX consults this .MAK file and automatically loads the modules listed in it.

Using DECLARE with Multiple Modules

The **DECLARE** statement has several important functions in BASIC. Using a **DECLARE** statement will do the following:

- Specify the sequence and data types of a procedure's parameters.
- Enable type-checking, which confirms, each time a procedure is called, that the arguments agree with the parameters in both number and data type.
- Identify a **FUNCTION** procedure's name as a procedure name, not a variable name.

QBX has its own system for automatically inserting the required **DECLARE** statements into your modules. The section "Checking Arguments with the DECLARE Statement" in Chapter 2, "SUB and FUNCTION Procedures," explains the features and limitations of this system.

Despite QBX's automatic insertion of the **DECLARE** statement, you may wish to create a separate include file that contains all the **DECLARE** statements required for a program. You can update this file manually as you add and delete procedures or modify your argument lists.

Note If you write your programs with a text editor (rather than in QBX) and compile from the command line, you must insert **DECLARE** statements manually.

Accessing Variables from Two or More Modules

You can use the **SHARED** attribute to make variables accessible at module level and within that module's procedures. If these procedures are moved to another module, however, these variables are no longer shared.

You could pass these variables to each procedure through its argument list. This may be inconvenient, though, if you need to pass a large number of variables.

One solution is to use **COMMON** statements, which enable two or more modules to access the same group of variables. The section "Sharing Variables with SHARED" in Chapter 2, "SUB and FUNCTION Procedures," explains how to do this.

Another solution is to use a **TYPE...END TYPE** statement to combine all the variables you wish to pass into a single structure. The argument and parameter lists then have to include only one variable name, no matter how many variables are passed.

If you are simply splitting up a program and its procedures into separate modules, either of these approaches works well. If, on the other hand, you are adding a procedure to a module (for use in other programs), you should avoid using a **COMMON** statement. Modules are supposed to make it easy to add existing procedures to new programs; **COMMON** statements complicate the process. If a procedure needs a large group of variables, it may not belong in a separate module.

Using Modules During Program Development

When you start a new programming project, you should first look through existing modules to see if there are procedures that can be reused in your new software. If any of these procedures aren't already in a separate module, you should consider moving them to one.

As your program takes shape, newly written procedures automatically become part of the program file (that is, the main module). You can move them to a separate module for testing or perhaps add them to one of your own modules along with other general-purpose procedures that are used in other programs.

Your program may need procedures written in other languages. (For example, MASM is ideal for direct interface with the hardware, FORTRAN has almost any math function you might want, Pascal allows the creation of sophisticated data structures, and C provides structured code and direct memory access.) These procedures are compiled and linked into a Quick library for use in your program. You can also write a separate module to test Quick library procedures in the same way you would test other procedures.

Compiling and Linking Modules

The end product of your programming efforts is usually a stand-alone executable file. You can create one in QBX by loading all of a program's modules, then choosing Make EXE File from the Run menu.

You can also compile modules from the command line using the BASIC Compiler (BC), then use LINK to combine the object code. Object files from code written in other languages can be linked at the same time.

Note

When you choose Make EXE File from QBX, all the module-level code and every procedure currently loaded is included in the executable file, whether or not the program uses this code. If you want your program to be as compact as possible, you must unload all unneeded module-level code and all unneeded procedures before compiling. The same rule applies when using BC to compile from the command line; all unused code should be removed from the files.

Quick Libraries

Although Microsoft Quick libraries are not modules, it is important that you understand their relationship to modules.

A Quick library contains nothing but procedures. These procedures can be written in BASIC or any other Microsoft language.

A Quick library contains only compiled code. (Modules contain BASIC source code.) The code in a Quick library can come from any combination of Microsoft languages. The chapter "Creating and Using Quick Libraries" explains how to create Quick libraries from object code and how to add new object code to existing Quick libraries.

Quick libraries have several uses:

- They provide an interface between BASIC and other languages.
- They allow designers to hide proprietary software. Updates and utilities can be distributed as Quick libraries without revealing the source code.
- They load faster and are usually smaller than modules. If a large program with many modules loads slowly, converting the modules other than the main module into a Quick library will improve loading performance.

Note, however, that modules are the easiest way to work on procedures during development because modules are immediately ready to run after each edit; you don't have to recreate the Quick library. If you want to put your BASIC procedures in a Quick library, wait until the procedures are complete and thoroughly debugged.

When a Quick library is created, any module-level code in the file it was created from is automatically included. However, other modules cannot access this code, so it just wastes space. Before converting a module to a Quick library, be sure that all module-level statements (except any error or event handlers and declarations that are used by the procedures) have been removed.

Note Quick libraries are not included in .MAK files and must be loaded with the /L option when you run QBX. A Quick library has the file extension .QLB. During the process of creating the Quick library, an object-module library file with the extension .LIB is created. This file contains the same code as the Quick library but in a form that allows it to be linked with the rest of the program to create a stand-alone application.

If you use Quick libraries to distribute proprietary code (data-manipulation procedures, for example), be sure to include the object-module library files (.LIB) so that your customers can create stand-alone applications that use these procedures. Otherwise, they will be limited to running applications within the QBX environment.

Creating Quick Libraries

You can create a Quick library of BASIC procedures with the Make Library command from the Run menu. The Quick library created contains every procedure currently loaded, whether or not your program calls it. (It also contains all the module-level code.) If you want the Quick library to be compact, be sure to remove all unused procedures and all unnecessary module-level code first.

You can create as many Quick libraries as you like, containing whatever combination of procedures you wish. However, only one Quick library can be loaded into QBX at a time. (You would generally create application-specific Quick libraries, containing only the procedures a particular program needs.) Large Quick libraries can be created by loading many modules, then choosing Make Library from the Run menu.

You can also compile one or more modules with the BC command and then link the object code files to create a Quick library. Quick libraries of procedures written in other languages are created the same way. In linking, you are not limited to one language; the object-code files from any number of Microsoft languages can be combined in one Quick library. Chapter 19, "Creating and Using Quick Libraries," explains how to convert object-code files (.OBJ) into Quick libraries.

Tips for Good Programming with Modules

You can use modules in any way you think will improve your program or help organize your work. The following suggestions are offered as a guide:

- Think and organize first.

  When you start on a new project, make a list of the operations you want to be performed by procedures. Then look through your own procedure library to see if there are any you can use, either as is or with slight modifications. Don't waste time "reinventing the wheel."

- Write generalized procedures with broad application.

  Try to write procedures that are useful in a wide variety of programs. Don't, however, make the procedure needlessly complex. A good procedure is a simple, finely honed tool.

  It is sometimes useful to alter an existing procedure to work in a new program. This might require modifying programs you've already written, but it's worth the trouble if the revised procedure is more powerful or has broader application.

- When creating your own procedure modules, keep logically separate procedures in separate modules.

  It makes sense to put string-manipulation procedures in one module, matrix-handling procedures in another, and data-communication procedures in a third. This arrangement avoids confusion and makes it easy to find the procedure you need.

Chapter 8
Error Handling

This chapter explains how to intercept and deal with errors that occur while a program is running. Using these methods, you can protect your program from such errors as opening a nonexistent file or trying to use the printer when it is out of paper.

After reading this chapter, you will know how to:

- Enable error trapping.
- Write an error-handling routine to process the trapped errors.
- Return control from an error-handling routine.
- Trap errors at the procedure and module level.
- Trap errors in programs composed of more than one module.

Why Use Error Handling?

Error handling is the process of intercepting and dealing with errors that occur at run time and cause your program to stop executing. A typical error would be if a stand-alone program attempted to save data on a disk that was out of space. For example:

```
OPEN "A:RESUME" FOR OUTPUT AS #1
PRINT #1, MyBusinessResume$
CLOSE #1
```

In this case, since there is no provision for error handling, BASIC issues the message `Disk Full` and terminates. The operator's data, stored in the string variable `MyBusinessResume$`, and all other data in memory, is lost.

To avoid this situation, you can use BASIC's error handling features to intercept errors and take action once they occur. For example, the full disk problem could be handled by making the following changes to the code (the details of this example are described in the next section):

```
ON ERROR GOTO Handler                    ' Turn on error trapping.
OPEN "A:RESUME" FOR OUTPUT AS #1         ' Write document to disk.
PRINT #1, MyBusinessResume$
CLOSE #1
END
' Branch here if an error occurs
Handler:
' Have operator get another disk if this one's full.
IF ERR = 61 THEN
   CLOSE #1
   KILL "A:RESUME"
   PRINT "Disk in drive A too full for operation."
   PRINT "Insert new formatted disk."
   PRINT "Press any key when ready"
   Pause$ = INPUT$(1)
   OPEN "A:RESUME" FOR OUTPUT AS #1
   RESUME
ELSE
   PRINT "Unanticipated error number";ERR ;"occurred."
   PRINT "Program Terminated."
   END
END IF
```

Your program can correct many kinds of operator errors using methods such as those shown in this example. You can also use error handling for program control and for gaining information about devices (monitors, printers, modems) that are connected to the computer.

How to Handle Errors

There are three steps necessary for handling errors:

1. Set the error trap by telling the program where to branch to when an error occurs.

   In the example in the preceding section, this is accomplished by the **ON ERROR GOTO** statement which directs the program to the label `Handler`.

2. Write a routine that takes action based on the specific error that has occurred—the error-handling routine.

   In the example in the preceding section, the `Handler` routine did this using a **IF THEN** statement. When it encountered a disk full error, it prompted the operator for a new disk. If that wasn't the solution to the error, it terminated.

3. Exit the error-handling routine.

   In the example in the preceding section, the **RESUME** statement was used to branch back to the line where the disk full error occurred. Data was then written to a new disk.

Details on how to perform these steps are given in the following sections.

Setting the Error Trap

There are two types of error traps—module level and procedure level. Both traps become enabled when BASIC executes the **ON [[LOCAL]] ERROR GOTO** statement. Once enabled, the module-level trap remains enabled for the duration of the program (or until you turn it off). The procedure-level trap is enabled only while the procedure containing it is active—that is until an **EXIT SUB** or **END SUB** statement is executed for that procedure. See the section "Procedure- vs. Module-Level Error Handling" later in this chapter for a discussion of when to use procedure- and module-level trapping.

To enable a module-level trap, use the **ON ERROR GOTO** *line* statement, where *line* is a line number or label identifying the first line of an error-handling routine. Place the statement in the code where you want error trapping enabled, usually at the beginning of the main module. As soon as BASIC executes this line, the trap is enabled.

To enable a procedure-level trap, add the **LOCAL** keyword to the statement so the syntax becomes **ON LOCAL ERROR GOTO** *line*. Place the statement in the procedure where you want error trapping enabled. As soon as BASIC executes this statement, the trap is enabled until an **EXIT SUB** or **END SUB** statement for this procedure is encountered.

Important The line number 0 when used in any **ON [[LOCAL]] ERROR GOTO** statement does not enable error handling. It has two other purposes: if it does not appear in an error-handling routine, it turns off error handling (see the section "Turning Off Error Handling" later in this chapter for details); if it appears in an error-handling routine, it causes BASIC to issue the error again.

Writing an Error-Handling Routine

This first statement in an error-handling routine contains the label or line number contained in the **ON [[LOCAL]] ERROR GOTO** *line* statement. For the example in this section, the label `PrintErrorHandler` is used.

Locate the error-handling routine in a place where it cannot be executed during normal program flow. A module-level error-handling routine is normally placed after the **END** statement. A procedure-level error-handling routine, is normally placed between the **EXIT SUB** and **END SUB** statements, as shown in the example in this section.

To find out which error occurred use the **ERR** function. It returns a numeric code for the program's most recent run-time error. (See Appendix D, "Error Messages," in the *BASIC Language Reference* for a complete list of the error codes associated with run-time errors.) By using **ERR** in combination with **SELECT CASE** or **IF THEN** statements, you can take specific action for any error that occurs.

Example In the following code, the `PrintList` procedure sends the array `List$()` to the printer. The **ERR** function is used to determine which message should be sent to the operator so the problem can be corrected.

```
SUB PrintList(List$())
ON LOCAL ERROR GOTO PrinterErrorHandler    ' Enable error trapping.
FOR I% = LBOUND(List$) to UBOUND(List$)
   LPRINT List$(I%)
NEXT I%
EXIT SUB
PrinterErrorHandler:    ' Error-handling routine
      ' Use ERR to determine which error
      ' caused the branch. Send out the appropriate message.
SELECT CASE ERR
   CASE 25
      ' 25 is the code for a Device fault error, which occurs
      ' if the printer is offline or turned off.
      PRINT "Printer offline or turned off"
   CASE 27
      ' 27 is the code for an Out of paper error:
      PRINT "Printer is out of paper."
   CASE 24
      ' 24 is the code for a Device timeout error, which
      ' occurs if the printer cable is disconnected.
      PRINT "Printer cable is disconnected."
   CASE ELSE
      PRINT "Unknown error."
      EXIT SUB
END SELECT

PRINT "Please correct the problem and"
PRINT "Press any key when ready."
Pause$ = INPUT$(1)
' Go try again after operator fixes the problem.
RESUME

END SUB
```

Exiting an Error-Handling Routine

To exit from an error-handling routine, use one of the statements shown in Table 8.1.

Table 8.1 Statements Used to Exit an Error-Handling Routine

Statement	Purpose
RESUME [[0]]	Returns to the statement that caused the error or the last call out of the error-handling procedure or module.
RESUME NEXT	Returns to the statement immediately following the one that caused the error or immediately following the last call out of the error-handling procedure or module.

Table 8.1 *Continued*

Statement	Purpose
RESUME *line*	Returns to the label or line specified by *line*.
ERROR ERR	Initiates a search of the invocation path for another error-handling routine.
ON ERROR GOTO 0	Initiates a search of the invocation path for another error-handling routine.

RESUME returns to the statement that caused the error or the last call out of the error-handling procedure or module. Use it to repeat an operation after taking corrective action.

The **RESUME NEXT** statement returns to the statement immediately following the one that caused the error or immediately following the last call out of the error-handling procedure or module. The difference between **RESUME** and **RESUME NEXT** is shown for a module-level error-handling routine in a single-module program in Figure 8.1.

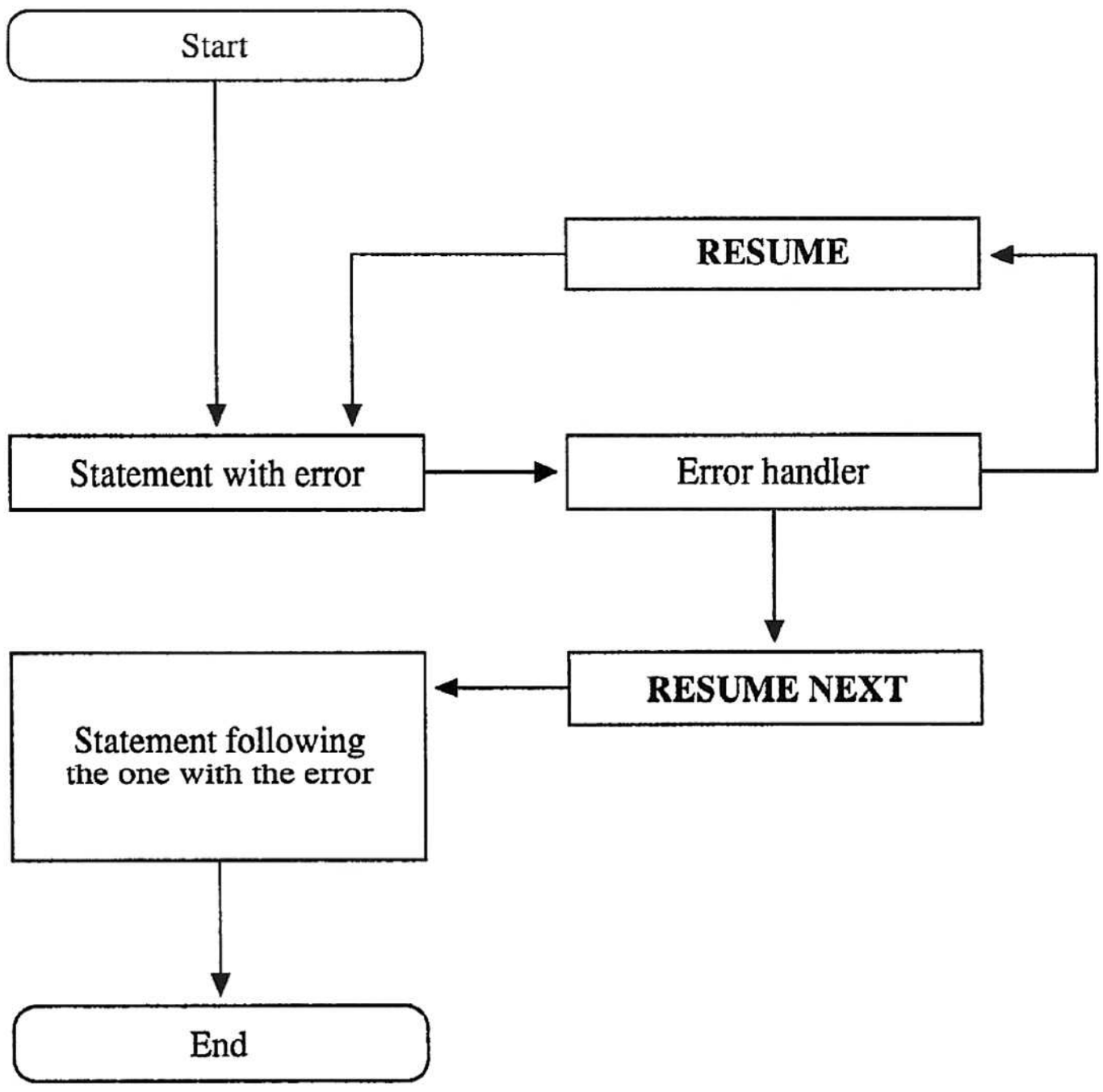


Figure 8.1 Program Flow with RESUME and RESUME NEXT

Use **RESUME NEXT** to ignore the error. For example, this code performs modulo 64K arithmetic by ignoring the overflow error.

```
ON ERROR GOTO Handler
' Add two hexadecimal numbers.
Result% = &H1234 + &HFFFF
PRINT HEX$( Result%)
END
Handler:
' If an overflow occurs, keep going.
RESUME NEXT
```

Sometimes, if an error occurs in a loop, you need to restart an operation rather than continue from where you left off with the **RESUME** statement. This would be true if a `Disk full` error occurs during execution of the following procedure which saves an array called `ClientNames$` on a floppy disk.

```
SUB SaveOnDisk (ClientNames$())
ON LOCAL ERROR GOTO Handler
Restart:
OPEN "A:CLIENTS.TXT" FOR OUTPUT AS #1
' Save array of client names to disk.
FOR I% = LBOUND(ClientNames$) TO UBOUND(ClientNames$)
PRINT #1, ClientNames$(I%)
NEXT I%
CLOSE #1
EXIT SUB
Handler:
' Have operator get another disk if this one's full.
IF ERR = 61 THEN
   CLOSE #1
   KILL "A:CLIENTS.TXT"
   PRINT "Disk Full.  Insert formatted disk"
   PRINT "Press any key when ready"
   Pause$ = INPUT$(1)
   RESUME Restart
ELSE
   ' Unanticipated error. Simulate the error & look for
   ' another handler.
   ERROR ERR
END IF

END SUB
```

In this case, the **RESUME** *line* statement was used, where *line* is any valid line number (except 0) or label. This allowed the error-handling routine to erase the incomplete file on the full disk, prompt the operator for a new disk and then return to the line labelled `Restart` to print the file from the beginning.

This last example also illustrates how to use **ERROR ERR**, the recommended way of handling an unanticipated error. When BASIC executes this statement, it simulates the occurrence of the error again, but this time the error occurs in the error-handling routine itself. In this case, BASIC searches back through the invocation path (if there is one) for another error-handling routine. If one is found, the program continues from that point. Otherwise, the appropriate error message is issued and the program terminates. The "invocation path" is a list of procedures that have been invoked in order to arrive at the program's current location. The invocation path is found on the stack and is also called the execution stack. (See the section "Unanticipated Errors" later in this chapter for details on how BASIC searches back through the invocation path.)

Note

To maintain compatibility with previous releases, this version of Microsoft BASIC allows you to use **ON [LOCAL] ERROR GOTO 0** in an error-handling routine to initiate a search of the invocation path. But because this same statement is used outside of error-handling routines to turn off error trapping, your coding will be clearer if you always use **ERROR ERR** within an error-handling routine. See "Turning Off Error Trapping" later in this chapter for examples of this other usage.

Procedure- Vs. Module-Level Error Handling

For many applications, procedure-level error handling is preferred. This is because procedures tend to be organized by task (video display driver, line printer, disk I/O, etc.), and errors are also task-related. Therefore program organization can be simpler and more straightforward when the two are grouped together.

Examples

This first example outlines how error trapping can be organized for several independent procedures; one for disk I/O, one for computations, and one for operator input. The module-level error-handling routine `LastResort` is used to take emergency action for an unanticipated error that the procedure-level error-handling routines don't deal with.

```
' Module-level code.
ON ERROR GOTO LastResort
' Main program executes here.
.
.
.
END

LastResort:
' Module-level error-handling routine
' It makes a last attempt to deal with unanticipated errors trapped in
' procedure-level error-handling routines.
.
.
.
END
```

```
SUB MathProcessor
ON LOCAL ERROR GOTO MathHandler
' Calculations are performed in this SUB.
.
.
.
EXIT SUB
MathHandler:
SELECT CASE ERR
   CASE 11
       ' Routine to handle divide by zero goes here.
       .
       .
       .
       RESUME NEXT
   CASE 6
        Routine to handle overflows goes here.
       .
       .
       .
       RESUME NEXT
   CASE ELSE
       ' Unanticipated error.
       ERROR ERR
END SELECT
END SUB

SUB DiskDriver
ON LOCAL ERROR GOTO DiskHandler
' Disk read and write performed in this SUB.
.
.
.
EXIT SUB
DiskHandler:
SELECT CASE ERR
CASE 72
    ' Routine to handle damaged disk goes here.
    .
    .
    .
    RESUME
CASE 71
    ' Routine to handle open drive door goes here.
    .
    .
    .
    RESUME
CASE 57
    ' Routine to handle internal disk drive problem goes here.
```

```
            .
            .
            .
            RESUME
        CASE 61
            ' Routine to handle full disk goes here.
            .
            .
            .
            RESUME
        CASE ELSE
            ' Unanticipated error.
            ERROR ERR
    END SELECT
    END SUB

    FUNCTION GetFileName
    ON LOCAL ERROR GOTO FileNameErrorHandler
    ' Code to get a filename from the operator goes here.
    .
    .
    .
    EXIT FUNCTION
    FileNameErrorHandler:
    SELECT CASE ERR
       CASE 64
            ' Routine to handle an illegal filename goes here.
            .
            .
            .
            RESUME
       CASE 76
            ' Routine to handle a non-existent path goes here.
            .
            .
            .
            RESUME
       CASE ELSE
    ' Unanticipated error—try searching invocation path.
            ERROR ERR
    END SELECT
    END FUNCTION
```

If you have several related procedures where the same kinds of errors may occur, it makes more sense to write one error-handling routine and put it at the module level. This is outlined in the following example which sends different kinds of data to the line printer.

```
ON ERROR GOTO HandleItAll
DECLARE SUB PrintNumericArray (StudentArray& ())
```

```
DECLARE SUB PrintStringArray (ClientArray$ ())
DIM Students& ( 1 to 100, 1 to 2), Clients$ (1 to 100, 1 to 2)
' Main-module-level code goes here. It acquires data from an operator
' and puts it in the previously dimensioned arrays.
.
.
.
CALL PrintNumericArray (Students&() )
CALL PrintStringArray (Clients&() )
END
HandleItAll:
' Branch here to handle an error in either SUB
SELECT CASE
   CASE 25
      ' Routine for handling printer off or disconnected goes here.
      .
      .
      .
      RESUME
   CASE 68
      ' Routine for handling printer being offline goes here.
      .
      .
      .
      RESUME
   CASE 27
      ' Routine for handling printer out of paper goes here.
      .
      .
      .
      RESUME
   CASE ELSE
      ERROR ERR
END SELECT

SUB PrintNumericArray (StudentArray& ())
' This prints a 2D numeric array of student numbers and test scores.
FOR I% = 1 to 100
LPRINT StudentArray&( I%, 1), StudentArray& (I%, 2)
NEXT I%
END SUB

SUB PrintStringArray (ClientArray$ ())
' This prints a 2D string array of client names and zip codes.
FOR I% = 1 to 100
LPRINT ClientArray$ ( I%, 1), ClienttArray$ (I%, 2)
NEXT I%
END SUB
```

Error Handling in Multiple Modules

Microsoft BASIC allows you to detect and handle errors that occur in multiple-module programs. To see how this works, try tracing through the following code. It begins in the main module where it handles a device I/O error. It then calls the procedure `Module2Sub` in the second module which calls the `Module3Sub` procedure in the third module. An `Out of memory` error occurs in `Module3Sub` which is handled at the module level. The program returns to `Module3Sub` which exits back to `Module2Sub` in the second module. Here a `Type mismatch` error occurs that is handled at the procedure level. The program then returns to `Module2Sub` which exits back to the main module and ends.

```
'=========================================================
'         MODULE #1 (MAIN)
'=========================================================
' Identify external procedure
DECLARE SUB Module2Sub ()
ON ERROR GOTO MainHandler
ERROR 57      ' Simulate occurrence of error 57
              ' (Device I/O error).
CALL Module2Sub
PRINT "Back in main-module after handling all errors."
END
MainHandler:
PRINT "Handling error"; ERR; "in main-module-level handler."
RESUME NEXT
' =========================================================
'         MODULE #2
' =========================================================
' Identify external procedure
DECLARE SUB Module3Sub ()

SUB Module2Sub
ON LOCAL ERROR GOTO Module2Handler
CALL Module3Sub
ERROR 13      ' Simulate a Type mismatch error.
EXIT SUB
```

```
Module2Handler:
PRINT "Handling error"; ERR; "in module 2 procedure-level handler."
RESUME NEXT

END SUB

' ========================================================
'         MODULE #3
' ========================================================
Module3Handler:
PRINT "Handling error"; ERR; "in module 3 module-level handler."
RESUME NEXT

SUB Module3Sub
ON ERROR GOTO Module3Handler
ERROR 7       ' Simulate an Out of memory error.
END SUB
```

Output

```
Handling error 57 in main-module-level handler
Handling error 7 in module 3 module-level handler
Handling error 13 in module 2 procedure-level handler
Back in main module after handling all errors.
```

In the preceding example, note the error-handling technique employed in the third module. The **ON ERROR GOTO** statement is used in a procedure without the **LOCAL** keyword. This is the only way to enable a module-level error-handling routine in a module other than the main module.

Unanticipated Errors

When an error occurs and there is no error-handling routine within the current scope of the program, BASIC searches the invocation path for an error-handling routine.

BASIC follows these steps when an unanticipated error is encountered:

1. BASIC searches each frame starting with the most recently invoked and continuing until an error-handling routine is found or the path is exhausted.
2. BASIC searches across module boundaries. Before crossing into another module, it searches the module-level code even if that code is not in the invocation path.
3. If no error-handling routine is found, the program terminates.
4. If an enabled error-handling routine is found, program execution continues in that error-handling routine. If the error-handling routine contains a **RESUME** or **RESUME NEXT** statement, the program continues as shown in Table 8.2.

Table 8.2 Program Continuation After BASIC Finds an Error-Handling Routine in the Invocation Path

Location of error-handling routine	Single module program	Multiple module program
Program continuation after RESUME		
Procedure-level	Returns to last executed statement in procedure where error-handling routine is found.	Returns to last executed statement in procedure where error-handling routine is found.
Module-level	Returns to statement where error occurred.	Returns to last executed statement in module where error-handling routine is found.
Program continuation after RESUME NEXT		
Procedure-level	Returns to statement following the last executed statement in procedure where error-handling routine is found.	Returns to statement following the last executed statement in procedure where error-handling routine is found.
Module-level	Returns to statement following the statement where the error occurred.	Returns to statement following the last executed statement in module where error-handling routine is found.

Examples

To understand how program flow is affected when unanticipated errors occur, consider the following example which performs this calculation: (3 * 3) + 2. It does this by passing the numbers 2 and 3 to a procedure called `Total`. This procedure calls `Square`. This second procedure squares the number 3 and returns to `Total` with the answer. `Total` then adds the number 2 to the answer and returns to the module level where the answer is printed.

```
DECLARE FUNCTION Total% (A%, B%)
DECLARE FUNCTION Square% (B%)
DEFINT A-Z
' Go do some calculations with the numbers 2 and 3.
Answer = Total(2, 3)
' Show us the results.
PRINT "(3 * 3) + 2 = "; Answer
END

FUNCTION Square (B)
' Find the square of B.
Square = B * B
END FUNCTION
```

```
FUNCTION Total (A, B)
ON LOCAL ERROR GOTO Handler
' Go square B, then add A to the result.
Result = Square(B)
Total = Result + A
EXIT FUNCTION
Handler:
' Ignore overflow errors.
IF ERR = 6 THEN
   RESUME NEXT
ELSE
   ERROR ERR
END IF
END FUNCTION
```

If no error occurs in our example we get the expected result:

```
(3 * 3) + 2 = 11
```

But suppose an error occurs in the `Square` function, as simulated by the following code:

```
FUNCTION Square (B)
' Simulate the occurrence of an overflow error.
ERROR 6
' Find the square of 3.
Square = B ^ 2
END FUNCTION
```

This time when `Square` is called, an error occurs before the calculation is made. BASIC finds no error-handling routine in this procedure, so it searches back through the invocation path for the closest available enabled error-handling routine. It finds one in `Total`, so it starts that error-handling routine. This error-handling routine has code for dealing with `ERROR 6` —an overflow error. In this case, it does a **RESUME NEXT**. Execution then continues in `Total` at the line following the call to `Square`, and the program proceeds as before. But when the result is printed, we get:

```
(3 * 3) + 2 = 2
```

This is because BASIC never returned to the `Square` procedure to make the calculation. This illustrates an important point: when BASIC encounters a **RESUME NEXT** statement in a procedure-level error-handling routine, it always returns to a statement in that procedure. This can be troublesome if you have a missing error-handling routine. In other words, your program may continue running, but it may not run as intended.

To see another example of how BASIC searches for error-handling routines, make the procedure-level error-handling routine in `Total` into a module-level error-handling routine so that the example looks like this:

```
DECLARE FUNCTION Total% (A%, B%)
DECLARE FUNCTION Square% (B%)
DEFINT A-Z
ON ERROR GOTO MainHandler
' Go do some calculations with the numbers 2 and 3.
Answer = Total(2, 3)
' Show us the results.
PRINT "(3 * 3) + 2 = "; Answer
END
MainHandler:
' Ignore overflow errors.
IF ERR = 6 THEN
   RESUME NEXT
ELSE
   ERROR ERR
END IF

FUNCTION Square (B)
' Simulate the occurrence of an error.
ERROR 6
' Find the square of B.
Square = B * B
END FUNCTION

FUNCTION Total (A, B)
' Go square B, then add A to the result.
Result = Square(B)
Total = Result + A
END FUNCTION
```

This time BASIC doesn't find an error-handling routine in the `Total` procedure so it looks further back and finds one at the module level. When the **RESUME NEXT** statement is executed in the `MainHandler` procedure, the program returns to the `Square` procedure and begins executing at the `Square = B * B` statement. Program flow then continues as in the original example that executed without an error. Thus the result this time is correct:

```
(3 * 3) + 2 = 11
```

The reason this last example works is because the error-handling routine was at the module level. This illustrates another important point: in a single module program, whenever BASIC encounters a **RESUME NEXT** statement in a module-level error-handling routine, it always returns to the statement directly following the one where the error occurred.

To see how unanticipated errors are handled in multiple-module programs, change the preceding example into two modules as follows:

```
'=============================================================
'            MODULE #1
'=============================================================
DEFINT A-Z
DECLARE FUNCTION Total (A, B)
DECLARE FUNCTION Square (B )
ON ERROR GOTO Handler
' Go do some calculations with the numbers 2 and 3.
Answer = Total ( 2, 3)
' Show us the results.
PRINT "(3 * 3) + 2 = "; Answer
END

Handler:
' Ignore overflow errors.
IF ERR = 6 THEN
   RESUME NEXT
ELSE
   ERROR ERR
END IF

'=============================================================
'            MODULE #2
'=============================================================
FUNCTION Total (A, B)
' Go square B, then add A to the result.
Result = Square( B)
Total = Result + A
END FUNCTION
FUNCTION Square ( B)
' Find the square of B.
ERROR 6
Square = B * B
END FUNCTION
```

Now when BASIC searches for an error-handling routine, it finds one in the first module. The **RESUME NEXT** causes the program to return to the line following the last executed line in that module. Therefore program execution continues with the **PRINT** statement and the result is:

```
(3 * 3) + 2 = 0
```

To see a final demonstration of program flow after dealing with an unanticipated error, move the module-level error-handling routine to the second module as shown here:

```
'=============================================================
'            MODULE #1
'=============================================================
DEFINT A-Z
DECLARE FUNCTION Total (A, B)
DECLARE FUNCTION Square (B )
ON ERROR GOTO Handler
' Go do some calculations with the numbers 2 and 3.
Answer = Total ( 2, 3)
' Show us the results.
PRINT "(3 * 3) + 2 = "; Answer
END
'=============================================================
'            MODULE #2
'=============================================================
Handler:
' Ignore overflow errors.
IF ERR = 6 THEN
   RESUME NEXT
ELSE
   ERROR ERR
END IF

FUNCTION Total (A, B)
' Enable the error handler at the module level
ON ERROR GOTO Handler
' Go square B, then add A to the result.
Result = Square( B)
Total = Result + A
END FUNCTION
FUNCTION Square ( B)
' Find the square of B.
ERROR 6
Square = B * B
END FUNCTION
```

This time BASIC finds an error-handling routine in the same module as where the error occurred. It therefore returns to the `Square` procedure and produces the correct answer:

```
(3 * 3) + 2 = 11
```

Guidelines for Complex Programs

Because BASIC makes such a thorough attempt to find missing error-handling routines, the following guidelines should be observed when writing complex programs with extensive operator interfaces:

- Write a "fail safe" error-handling routine for the main module of your program application. This will be executed if a search for an error-handling routine is unsuccessful and winds up back at the main module level. The error-handling routine should make an attempt to save the operator's data before the program terminates.
- Use procedure-level error-handling routines wherever possible to deal with anticipated errors. Errors caught in a procedure can usually be corrected more easily there.
- Put an **ERROR ERR** statement in all procedure-level error-handling routines and in all module-level error-handling routines outside of the main module in case there is no code in the error-handling routine to deal with a specific error. This lets your program try to correct the error in other error-handling routines along the invocation path.
- Use the **STOP** statement to force termination if you don't want a previous procedure or module to trap the error.

Errors Within Error- and Event-Handling Routines

An error occurring within an error-handling routine is treated differently depending on whether the error-handling routine is for an error or an event:

- If an error occurs in an error-handling routine, BASIC begins searching the invocation path using the principles demonstrated in the section "Unanticipated Errors" earlier in this chapter.
- If an error occurs in an event-handling routine, including any procedure called by the event-handling routine, BASIC also searches the invocation path, but only as far back as the event frame. If it can't find an error-handling routine in that part of the invocation path, or in the module-level code, it terminates.

Delayed Error Handling

At times you may need to detect errors but not handle them until a certain section of code is finished executing. Do this with the **ON ERROR RESUME NEXT** statement which tells BASIC to record the error but not interrupt the program.

You then can write a routine for dealing with the errors that can be executed whenever it is convenient. The routine can tell if an error occurred by the value of **ERR**. If **ERR** is not zero, an error has occurred and the error-handling routine can take action based on the value of **ERR** as shown by the following example:

```
CONST False = 0, True = NOT False
' Don't let an error disrupt the program.
ON ERROR RESUME NEXT
Condition% = False
DO UNTIL Condition% = True
' A long calculation loop that exits when the
' variable Condition becomes true.
.
.
.
LOOP
' Now see if an error occurred and if so take action.

SELECT CASE ERR
   CASE 0
   ' No error—Don't send a message. Continue with program.
      Goto MoreProgram
   ' Find out which error it is and deal with it.
   CASE 6
      PRINT "An overflow has occurred"
   CASE 11
      PRINT "You have attempted to divide by zero"
END SELECT
PRINT "Enter 'I' to ignore, anything else to quit"; S$
S$ = input(1)
IF UCASE$(S$) <> "I" THEN END
```

```
MoreProgram:
' Program continues here after checking for errors.
.
.
.
END
```

There are two points to remember when doing this kind of error handling:

- The routine that detects and deals with the error (ERR) is different from the error-handling routines we have been discussing—it does not use any of the **RESUME** statements.
- The error number contained in **ERR** is the number of the most recent error. Any other errors that occurred earlier in the preceding loop are lost.

Another reason for using **ON ERROR RESUME NEXT** is to tailor the error handling to each statement, or a group of related statements, in your code rather than having a single error-handling routine per procedure. This is outlined by the following example which opens a file, converts it to ASCII text, and sends it to the line printer:

```
SUB ConvertToASCII (Filename$)
ON LOCAL ERROR RESUME NEXT
' Try to open the specified file. Correct errors if they occur.
OPEN Filename$ FOR INPUT AS #1
IF ERR < > 0 THEN
   OpenErrorHandler:
   ' Routine to identify and deal with file-open errors.
   .
   .
   .
END IF
FOR Counter% = 1 to LOF(1)
   ' Read in a character and correct errors if they occur.
   S$ = INPUT$(1, #1)
   IF ERR < > THEN
      InputError-handling routine:
      ' Routine to identify and deal with file input errors.
      .
      .
      .
END IF
ASCII% = ASC( S$) AND &H7F
' Print a character and correct errors if they occur.
LPRINT CHR$(Ascii%);
IF ERR < > THEN
    ' Routine to identify and deal with printer errors.
    .
    .
    .
```

```
NEXT Counter%
PrinterErrorHandler:
END SUB
```

Turning Off Error Handling

To turn off error handling, use the **ON [LOCAL] ERROR GOTO 0** statement. Once BASIC executes this statement, errors are neither trapped nor detected. If the **LOCAL** keyword is used, then error handling is turned off only within the procedure where the statement appears. Otherwise error handling is turned off for the current module error-handling routine.

Important The only place you cannot turn off error handling is in the error-handling routine itself. If BASIC encounters an **ON ERROR GOTO 0** statement there, it will treat it the same as an **ERROR ERR** statement and begin searching back through the invocation path for another error-handling routine.

Example The following example turns off error handling in the procedure `DemoSub`.

```
SUB DemoSub
ON ERROR GOTO SubHandler
' Error trapping is enabled.
' Errors need to be caught and corrected here.
.
.
.
ON ERROR GOTO 0
' Error trapping is turned off here because it's not needed.
.
.
.
ON ERROR GOTO SubHandler
' Error trapping is enabled again.
.
.
.
EXIT SUB
SubHandler:
' Error-handling routine goes here.
.
.
.
RESUME
END SUB
```

Additional Applications

Besides handling operator errors, you can also use error handling for program control. Although not recommended as a general rule, occasionally it is a convenient method. As an example of this, consider the following code which gets a number from the operator and returns the arc tangent. Because the arc tangent of zero or any multiple of π produces a `Division by zero` error, an error-handling routine is employed to ignore the result of the computation when the operator enters one of those values.

```
CONST False = 0, True = NOT False
INPUT "Enter a number";Number
ON ERROR GOTO Handler
NoError = True
ArcTangent = 1 / (TAN (Number))
PRINT "The arctangent of "; Number; "is ";
' If the answer is a real number then print it.
IF NoError THEN
   PRINT ArcTangent
' Otherwise tell the operator the answer is undefined (infinite).
ELSE PRINT "undefined"
END IF
END

Handler:
NoError = False
RESUME NEXT
```

Another use for error handling is to gain information, unavailable with other techniques, about the computer on which your application is running. For example, the following procedure `Adapter`, tells the operator which display adapter is installed, based on the errors produced by various **SCREEN** statements.

```
DEFINT A-Z

SUB Adapter
ON LOCAL ERROR GOTO Handler
CONST False = 0, True = NOT FALSE
' Use an array to keep track of our test results.
DIM Mode( 1 to 13)

' Try screen modes and see which work.
FOR ModeNumber = 1 to 13
   ' Assume the test works unless you get an error.
   Mode (ModeNumber) = True
   SCREEN ModeNumber
NEXT ModeNumber

' Reset the screen after testing it.
SCREEN 0, 0
WIDTH 80
```

```
' Using test results figure out which adapter is out there.
' Tell operator which one he has.
' (See tables in SCREEN statement section of BASIC Language Reference
' to understand why this logic works.)
PRINT "You have a";
IF Mode(13) THEN
   IF Mode(7) THEN
      PRINT "VGA";
      UsableModes$ = "0-2, 7-13."
   ELSE
      PRINT "MCGA";
      UsableModes$ = "0-2, 11 & 13."
   END IF
ELSE
   IF Mode(7) THEN
      PRINT "EGA";
      UsableModes$ = "0-2, 7-10."
   ELSE
      IF Mode(3) THEN
         PRINT "Hercules";
         UsableModes$ = "3."
      END IF
      IF Mode(4) THEN
         PRINT "Olivetti";
         UsableModes$ = "4."
      END IF
      IF Mode(1) THEN
         PRINT "CGA";
         UsableModes$ = "0-2."
      ELSE
         PRINT "MDPA";
         UsableMode$ = "0."
      END IF
   END IF
END IF
PRINT "Graphics card that supports screen mode(s) "; UsableModes$
EXIT SUB

' Branch here when test fails and change text result.
Handler:
Mode (ModeNumber) = False
RESUME NEXT
END SUB
```

Output with VGA Adapter

```
You have a VGA Graphics card that supports screen modes(s) 0-2, 7-13.
```

Note The list of screen modes produced by the preceding example may be incomplete for some non-IBM VGA and EGA adapters.

Trapping User-Defined Errors

There are many error codes that are not used by Microsoft BASIC. By using an unused error code in an **ERROR** statement, you can let BASIC's error handling logic control program flow for special error conditions you anticipate.

Example The following simplified example uses an error-handling routine in a procedure called `CertifiedOperator` to control the operator's access to a network. The procedure checks to see that the operator's password is "Swordfish" and that the first four numbers of the account number are 1234. If the operator doesn't enter the correct information after three attempts, the procedure returns false and the network connection is not made.

```
'PASSWRD.BAS
CONST False = 0, True = NOT False
DECLARE FUNCTION CertifiedOperator%

IF CertifiedOperator = False THEN
   PRINT "Connection Refused."
   END
END IF

PRINT "Connected to Network."
' Main program continues here.
.
.
.
END

FUNCTION CertifiedOperator%
ON LOCAL ERROR GOTO Handler
' Count the number of times the operator tries to sign on.
Attempts% = 0

TryAgain:
' Assume the operator has valid credentials
CertifiedOperator = True
' Keep track of bad entries
Attempts% = Attempts% + 1
IF Attempts% > 3 then ERROR 255
' Check out the operator's credentials
INPUT "Enter Account Number"; Account$
IF LEFT$ (Account$, 4) <> "1234" THEN ERROR 200
```

```
INPUT "Enter Password"; Password$
IF Password$ <> "Swordfish" THEN ERROR 201
EXIT SUB

Handler:
SELECT CASE
   ' Start over if account number doesn't have "1234" in it.
   CASE 200
      PRINT "Illegal account number. Please re-enter both items."
      RESUME TryAgain
   ' Start over if the password is wrong.
   CASE 201
      PRINT "Wrong password. Please re-enter both items."
      RESUME TryAgain
   ' Return false if operator makes too many mistakes.
   CASE 255
      CertifiedOperator% = FALSE
      EXIT SUB
END SELECT

END SUB
```

Compiling from the Command Line

When compiling from the command line, you must use one or both of these error-handling options:

- Use /E if your code contains:

  ON [[LOCAL]] ERROR GOTO

  RESUME *line*

- Use /X if your code contains:

  RESUME [[0]]

  RESUME NEXT

  ON [[LOCAL]] ERROR RESUME NEXT

Note You will get compilation errors if you do not use /E and /X in the preceding situations.

Chapter 9
Event Handling

This chapter explains how to detect and respond to events that occur while your program is running. This process is called "event handling." After reading this chapter, you will know how to:

- Specify an event to trap and enable event handling.
- Write a routine to process the trapped event.
- Return control from an event-handling routine.
- Write a program that traps any keystroke or combination of keystrokes.
- Trap music events and user-defined events.
- Trap events in programs composed of more than one module.

Event Trapping Vs. Polling

Many times during program execution, an event occurs which requires the program to suspend normal operation and take some action. The event could be the operator pressing the Ctrl+Break key combination, the arrival of data at the communications port, or the ending of a phrase of music playing in the background of a computer game.

There are two ways to detect these events: polling and event trapping. To understand the difference, imagine we are in a loop which will continue forever unless the operator presses Ctrl+C. To detect this with polling, you write code that must be executed repeatedly. In this example, the **INKEY$** function is performed every time the loop is executed:

```
DO
' Normal flow of program occurs here.
.
.
.
' Loop until operator presses Ctrl+C (ASCII 03).
LOOP UNTIL INKEY$ = CHR$(3)
' Program interrupted by the operator stops here.
END
```

Although polling works for the preceding example, it can degrade performance to check for events this way. And if the loop is too big, you might miss events that occur too quickly.

A better alternative for many cases is to use event trapping. This scheme allows BASIC to do the detection on an interrupt basis and redirect the program as soon as the interrupt is detected. For the preceding scenario, the event trap would look like the following (the details of this example are described in the next section):

```
' Define the key depression to look for (Ctrl+C),
' where to go when it's pressed,
' and turn on event trapping.
KEY 15, CHR$(4) + CHR$(46)
ON KEY (15) GOSUB Handler
KEY (15) ON
DO
' Normal program flow occurs here.
.
.
LOOP
' Branch here when Ctrl+C is pressed.
Handler:
END
```

How to Trap Events

There are three steps necessary for event trapping:

1. Set the trap by telling the program where to branch to when a specific event occurs.

   In the preceding example, this is accomplished by the `ON KEY (15) GOSUB` statement which directs the program to the label `Handler`.

2. Turn on event trapping for the particular event you want.

   In the preceding example, the `KEY (15) ON` does this.

3. Write a routine that takes action based on the specific event that has occurred—the event-handling routine.

   In the preceding example, the action was very simple: the program is terminated with the **END** statement. At other times, you may need to go back to the main program. In that case, you put a **RETURN** statement at the end of the event-handling routine.

If you are trapping a predefined event (such as the pressing of one of the function keys), only the three preceding steps are required. Otherwise, you need another line of code to define the event that is to be trapped. This was done in the preceding example with the **KEY** statement which informed BASIC that the key we were looking for was Ctrl+C.

Examples of trapping predefined and user-defined events can be found throughout this chapter.

Where to Put the Event-Handling Routine

The event-handling routine must be in the module-level code. This is the only place where BASIC will look for it. If you accidentally put it in a BASIC procedure, you will get a `Label not found` error at run time.

Usually the event-handling routine goes after the **END** statement as in this sample fragment:

```
END
SampleHandler:
PRINT "You have pressed the F1 Key."
PRINT "It caused a branch to the event-handling routine."
RETURN
```

The event-handling routine is located here so that it does not get executed during normal program flow. You could also put it above the **END** statement and skip around it with the **GOTO** statement, but putting the event-handling routine after **END** is the better programming practice.

Events that BASIC Can Trap

The following table shows the BASIC events you can trap and the statements used to trap them. Examples of trapping keystrokes, music events and user-defined events are found in the following sections.

BASIC statement	Event trapped
COM(*n*%)	Data from one of the serial ports (1 or 2) appearing in the communications buffer (the intermediate storage area for data sent to or received from the serial port).
KEY(*n*%)	The user pressing the given key.
PEN	The user activating the light pen.
PLAY(*n*)	The number of notes remaining to be played in the background dropping below *n*.
STRIG(*n*%)	The user squeezing the trigger of the joystick.
TIMER(*n*&)	The passage of *n*& seconds.
UEVENT	A user-defined event.

Trapping Preassigned Keystrokes

To detect any of the following preassigned keystrokes and route program control to a key-press routine, you need both of the following statements in your program:

ON KEY(*n%*) **GOSUB** *line*
KEY(*n%*) **ON**

Here, the *n%* value corresponds to the following keys:

Value	Key
1–10	Function keys F1-F10 (sometimes called "soft keys")
11	The Up Arrow key
12	The Left Arrow key
13	The Right Arrow key
14	The Down Arrow key
15–25	User-defined keys
30	The F11 function key (101-key keyboard only)
31	The F12 function key (101-key keyboard only)

The following two lines cause the program to branch to the `KeySub` routine each time the F2 function key is pressed:

```
ON KEY(2) GOSUB KeySub
KEY(2) ON
.
.
.
```

The following four lines cause the program to branch to the `DownKey` routine when the Down direction key is pressed and to the `UpKey` routine when the Up Arrow key is pressed:

```
ON KEY(11) GOSUB UpKey
ON KEY(14) GOSUB DownKey
KEY(11) ON
KEY(14) ON
.
.
.
```

Important For compatibility with previous versions of BASIC, the **ON KEY** (n) **GOSUB** statement traps all function key depressions whether or not they are shifted. This means that you cannot use event trapping to distinguish between, for example, F1, Ctrl+F1, Alt+F1 and Shift+F1.

Trapping User-Defined Keystrokes

In addition to providing the preassigned key numbers 1–14 (plus 30 and 31 with the 101-key keyboard), BASIC allows you to assign the numbers 15–25 to any of the remaining keys on the keyboard. The key can be any single key such as the lowercase "s," or it can be a key combination, such as Ctrl+Z, as explained in the next two sections.

Defining and Trapping a Non-Shifted Key

To define and trap a single key, use these three statements:

KEY *n%*, **CHR$(0)** + **CHR$**(*code%*)
ON KEY(*n%*) **GOSUB** *line*
KEY(*n%*) **ON**

Here, *n%* is a value from 15 to 25, and *code%* is the scan code for that key. (See Appendix A in the *BASIC Language Reference* for a listing of keyboard scan codes.) The `CHR(0)` function, used in the first line, tells BASIC that the trapped key is a single key.

The following example causes the program to branch to the `TKey` routine each time the user presses the lowercase "t":

```
' Define key 15 (the scan code for "t" is decimal 20):
KEY 15, CHR$(0) + CHR$(20)

' Define the trap (where to go when "t" is pressed):
ON KEY(15) GOSUB TKey
KEY(15) ON                          ' Turn on detection of key 15.

PRINT "Press q to end."
DO                                  ' Idle loop: wait for user to
LOOP UNTIL INKEY$ = "q"             ' press "q", then exit.
END

TKey:                               ' Key-handling routine
   PRINT "Pressed t."
RETURN
```

Defining and Trapping a Shifted Key

You can also set a trap for key combinations. A key is a combination if it is pressed simultaneously with one or more of the special keys Shift, Ctrl, or Alt or if pressed after toggling on the keys NumLock or Caps Lock.

This is how to trap the following key combinations:

KEY *n%*, **CHR$**(*keyboardflag%*) + **CHR$**(*code%*)
ON KEY(*n%*) **GOSUB** *line*
KEY(*n%*) **ON**

Here, *n%* is a value from 15 to 25, *code%* is the scan code for the primary key, and *keyboardflag%* is the sum of the individual codes for the special keys pressed, as shown in the following table:

Key	Code for *keyboardflag*
Shift	1, 2, or 3 Key trapping assumes the left and right Shift keys are the same, so you can trap the Shift key with 1 (left), 2 (right), or 3 (left + right)
Ctrl	4
Alt	8
NumLock	32
Caps Lock	64
Any extended key on the 101-key keyboard (in other words, a key such as Left or Del that is not on the numeric keypad)	128

For example, the following statements turn on trapping of Ctrl+S. Note these statements are designed to trap the Ctrl+S (lowercase) and Ctrl+Shift+S (uppercase) key combinations. To trap the uppercase S, your program must recognize capital letters produced by holding down the Shift key, as well as those produced when the Caps Lock key is active, as shown here:

```
' 31 = scan code for S key
' 4 = code for Ctrl key
KEY 15, CHR$(4) + CHR$(31)    ' Trap Ctrl+S.

' 5 = code for Ctrl key + code for Shift key
KEY 16, CHR$(5) + CHR$(31)    ' Trap Ctrl+Shift+S.

' 68 = code for Ctrl key + code for CAPSLOCK
KEY 17, CHR$(68) + CHR$(31)   ' Trap Ctrl+CAPSLOCK+S.

ON KEY (15) GOSUB CtrlSTrap   ' Tell program where to
ON KEY (16) GOSUB CtrlSTrap   ' branch (note: same
ON KEY (17) GOSUB CtrlSTrap   ' routine for each key).

KEY (15) ON                   ' Activate key detection for
KEY (16) ON                   ' all three combinations.
KEY (17) ON
.
.
.
```

The following statements turn on trapping of Ctrl+Alt+Del:

```
' 12 = 4 + 8 = (code for Ctrl key) + (code for Alt key)
' 83 = scan code for Del key
KEY 20, CHR$(12) + CHR$(83)
ON KEY(20) GOSUB KeyHandler
KEY(20) ON
.
.
.
```

Note in the preceding example that the BASIC event trap overrides the normal effect of Ctrl+Alt+Del (system reset). Using this trap in your program is a handy way to prevent the user from accidentally rebooting while a program is running.

If you use a 101-key keyboard, you can trap any of the keys on the dedicated keypad by assigning the string as to any of the *n*% values from 15 to 25:

CHR$(128) + CHR$(*code%***)**

Example The following example shows how to trap the Left Arrow keys on the dedicated cursor keypad and the numeric keypad.

```
' 128 = keyboard flag for keys on the
' dedicated cursor keypad
' 75 = scan code for Left Arrow key

KEY 15, CHR$(128) + CHR$(75)  ' Trap Left key on
ON KEY(15) GOSUB CursorPad    ' the dedicated
KEY(15) ON                    ' cursor keypad.

ON KEY(12) GOSUB NumericPad   ' Trap Left key on
KEY(12) ON                    ' the numeric keypad.

DO: LOOP UNTIL INKEY$ = "q"   ' Start idle loop.
END

CursorPad:
   PRINT "Pressed Left key on cursor keypad."
RETURN

NumericPad:
   PRINT "Pressed Left key on numeric keypad."
RETURN
```

Trapping Music Events

When you use the **PLAY** statement to play music, you can choose whether the music plays in the foreground or in the background. If you choose foreground music (which is the default) nothing else can happen until the music finishes playing. However, if you use the **MB** (Music Background) option in a **PLAY** music string, the tune plays in the background while subsequent statements in your program continue executing.

The **PLAY** statement plays music in the background by feeding up to 32 notes at a time into a buffer, then playing the notes in the buffer while the program does other things. A "music trap" works by checking the number of notes currently left to be played in the buffer. As soon as this number drops below the limit you set in the trap, the program branches to the first line of the specified routine.

To set a music trap in your program, you need the following statements:

ON PLAY(*queuelimit%*) **GOSUB** *line*
PLAY ON
PLAY "MB"
.
.
.
PLAY *commandstring$*
.
.
.

Here, *queuelimit%* is a number between 1 and 32. For example, this fragment causes the program to branch to the `MusicTrap` routine whenever the number of notes remaining to be played in the music buffer goes from eight to seven:

```
ON PLAY(8) GOSUB MusicTrap
PLAY ON
.
.
PLAY "MB" ' Play subsequent notes in the background.
PLAY "o1 A# B# C-"
.
.
.
MusicTrap:
   .   ' Routine to play addition notes in the background.
   .
   .
RETURN
```

Important A music trap is triggered only when the number of notes goes from *queuelimit%* to *queuelimit* – 1. For example, if the music buffer in the preceding example never contained more than seven notes, the trap would never occur. In the example, the trap happens only when the number of notes drops from eight to seven.

Example You can use a music-trap routine to play the same piece of music repeatedly while your program executes, as shown in the following example:

```
' Turn on trapping of background music events:
PLAY ON

' Branch to the Refresh subroutine when there are fewer than
' two notes in the background music buffer:
ON PLAY(2) GOSUB Refresh

PRINT "Press any key to start, q to end."
Pause$ = INPUT$(1)

' Select the background music option for PLAY:
PLAY "MB"

' Start playing the music, so notes will be put in the
' background music buffer:
GOSUB Refresh

I = 0

DO

   ' Print the numbers from 0 to 10,000 over and over until
   ' the user presses the "q" key. While this is happening,
   ' the music will repeat in the background:
   PRINT I
   I = (I + 1) MOD 10001
LOOP UNTIL INKEY$ = "q"

END

Refresh:

   ' Plays the opening motive of
   ' Beethoven's Fifth Symphony:
   Listen$ = "t180 o2 p2 p8 L8 GGG L2 E-"
   Fate$ = "p24 p8 L8 FFF L2 D"
   PLAY Listen$ + Fate$
   RETURN
```

Trapping a User-Defined Event

This section uses assembly language examples and calls to the DOS operating system. You may want to skip it if you are unfamiliar with these items.

Trapping a user-defined event involves writing a non-BASIC routine, such as in Microsoft Macro Assembler (MASM) or C, to define the event and inform BASIC when the event occurs. Once this is done, and the routine is installed, program flow continues much as in the preceding examples but uses these statements instead:

ON UEVENT GOSUB *line*
UEVENT ON

Example As an example of trapping a user-defined event, suppose you have a special task that needs to be done every 4.5 seconds. The following code accomplishes this, using the system timer chip which provides an interrupt to the CPU 18.2 times per second. The interrupt is DOS number 1CH. The address where DOS expects to find the far pointer to the interrupt service routine is 0:70H.

The code requires three MASM routines. The first one, `SetInt`, informs DOS that a new interrupt service routine `ErrorHandler` is to be executed whenever interrupt 1CH occurs.

The second routine, `EventHandler`, is called 18.2 times per second. It increments a counter. After 82 interrupts (4.5 seconds) it calls the `SetUevent` routine which is loaded from the BASIC main library during compilation. The routine sets a flag indicating that a user event has occurred. When BASIC encounters the flag, it causes the BASIC program to branch to the `SpecialTask` routine indicated in the **ON UEVENT GOSUB** statement.

The third routine, `RestInt`, restores the original service routine for the interrupt when the BASIC program terminates.

```
.model  medium, basic          ; Stay compatible
.data                          ; with BASIC.
.code
SetInt proc   uses ds          ; Get old interrupt vector
mov  ax, 351CH                 ; and save it.
   int  21h
   mov  word ptr cs:OldVector, bx
   mov  word ptr cs:OldVector + 2, es

   push  cs                    ; Set the new
   pop  ds                     ; interrupt vector
   lea  dx, Eventhandler       ; to the address
   mov  ax, 251CH              ; of our service
   int  21H                    ; routine.
   ret
SetIntendp
```

```
public  EventHandler            ; Make the following routine public
EventHandler  proc              ; for debugging.
   extrn  SetUevent: proc       ; Define BASIC library routine.
   push  bx
   lea  bx, cs:TimerTicks       ; See if 4.5 secs have passed.
   inc  byte ptr cs:[bx]
   cmp  byte ptr cs:[bx], 82
   jnz  Continue
   mov  byte ptr cs:[bx], 0     ; if true, reset counter,
   push   ax                    ; save registers, and
   push   cx                    ; have BASIC
   push   dx                    ; set the user
   push   es                    ; event flag.
   call   SetUevent
   pop  es
   pop  dx                      ; Restore registers.
   pop  cx
   pop  ax
Continue:
   pop  bx
   jmp  cs:OldVector            ; Continue on with the
                                ; old service routine.

TimerTicks  db  0               ; Keep data in code segment
OldVector  dd  0                ; where it can be found no
                                ; matter where in memory the
EventHandler endp               ; interrupt occurs.

RestInt   proc   uses ds        ; Restore the old
   lds  dx, cs:OldVector        ; interrupt vector
   mov   x, 251CH               ; so things will
   int  21h                     ; keep working when
   ret                          ; this BASIC program is
RestInt          endp           ; finished.
   end
```

The BASIC program shown here provides an outline of how our special task is performed using the **UEVENT** statement. The program first installs the interrupt, sets the path to the BASIC event-handling routine and then enables event trapping.

```
' Declare external MASM procedures.
DECLARE SUB SetInt
DECLARE SUB RestInt
```

```
' Install new interrupt service routine.
CALL SetInt

' Set up the BASIC event handler.
ON UEVENT GOSUB SpecialTask
UEVENT ON

DO
' Normal program operation occurs here.
' Program ends when any key is pressed.
LOOP UNTIL INKEY$ <> ""

' Restore old interrupt service routine before quitting.
CALL RestInt

END

' Program branches here every 4.5 seconds.
SpecialTask:
' Code for performing the special task goes here, for example:
PRINT "Arrived here after 4.5 seconds."
RETURN
```

Generating Smaller, Faster Code

Event trapping adds execution time and code length to a BASIC program. To make your programs smaller and faster, you can turn off event trapping in sections where it is unnecessary. Do this with the **EVENT OFF** statement, as shown in the following example. When **EVENT OFF** is encountered by the compiler, it stops generating code to trap events, but the events will still be detected. When the compiler encounters the **EVENT ON** statement, code is inserted to re-enable the traps. Any previously detected event, and any newly occurring event, will cause the program to branch to the appropriate event-handling routine.

```
ON KEY (1) GOTO Handler1
ON KEY (2) GOTO Handler2
KEY (1) ON
KEY (2) ON
' All events are trapped here.
.
.
.
```

```
EVENT OFF
' Events are still detected but no longer trapped.
' Code generated for these statements is smaller and faster.
.
.
.
EVENT ON
' Events are trapped again including previously detected ones.
.
.
.
END
Handler1:
' F1 key event-handling routine goes here.
RETURN
Handler2:
' F2 key event-handling routine goes here.
RETURN
```

Turning Off and Suspending Specific Event Traps

If you want to selectively turn off certain event traps and leave others on, use the *event* **OFF** statement. The *event* occurring after an *event* **OFF** statement has been executed is then ignored. This is shown by the following example where the F1 key trap is turned off, but the trap for F2 is left on:

```
ON KEY (1) GOTO Handler1
ON KEY (2) GOTO Handler2
KEY (1) ON
KEY (2) ON
' Both key traps are turned on here.
.
.
.
KEY (1) OFF
' The F1 trap is turned off and ignored here, but we still trap F2.
.
.
.
```

```
KEY (1) ON
' Now we can trap them both again.
.
.
.
END
Handler1:
' F1 key event-handling routine goes here.
RETURN
Handler2:
' F2 key event-handling routine goes here.
RETURN
```

Sometimes you need to suspend event trapping, without turning it off. This allows you to record events that occur and take action on them after a specific time period has elapsed.

To suspend event trapping, use the *event* **STOP** statement as demonstrated in the next example. In the following example, after the *event* **STOP** statement is encountered, if the timer event occurs, there is no branch to the event-handling routine. However, the program remembers that the event occurred, and as soon as trapping is turned back on with *event* **ON**, it immediately branches to the `ShowTime` routine.

```
' Once every minute (60 seconds),
' branch to the ShowTime routine:
ON TIMER(60) GOSUB ShowTime

' Activate trapping of the 60-second event:
TIMER ON
.
.
.
TIMER STOP   ' Suspend trapping.
   ' A sequence of lines you don't want interrupted,
   ' even if 60 or more seconds elapse.
.
.
.
TIMER ON     ' Reactivate trapping.
   ' If a timer event occurs above, it will now
   ' be handled by the ShowTime routine.
.
.
.
END
```

```
ShowTime:

   ' Get the current row and column position of the cursor,
   ' and store them in the variables Row and Column:
   Row    = CSRLIN
   Column = POS(0)

   ' Go to the 24th row, 20th column, and print the time:
   LOCATE 24, 20
   PRINT TIME$

   ' Restore the cursor to its former position
   ' and return to the main program:
   LOCATE Row, Column
RETURN
```

Events Occurring Within Event-Handling Routines

If an event occurs during an event-handling routine, and the event trap is on for the new event, then the program branches to the event-handling routine for this new event. For instance, in the first example in the preceding section, if the F2 key is pressed while `Handler1` is executing, the program branches to `Handler2`. When `Handler2` is finished, the program returns to `Handler1` and then goes back to the main program.

The only time that branching doesn't occur in a event-handling routine is if both events are the same. In this case branching cannot occur because event-handling routines execute an implicit *event* **STOP** statement for a given event whenever program control is in the routine. This is followed by an implicit *event* **ON** for that event when program control returns from the routine.

For example, if a key-handling routine is processing a keystroke, trapping the same key is suspended until the previous keystroke is completely processed by the routine. If the user presses the same key during this time, this new keystroke is remembered and trapped after control returns from the key-handling routine.

Event Trapping Across Modules

Events whose traps are turned on in one module are detected and trapped in any module that is running. This is demonstrated in the following program where a trap set for the F1 function key in the main module is triggered even when program control is in the other module.

```
' ========================================================
'         MODULE
' ========================================================
ON KEY (1) GOSUB GotF1Key
KEY (1) ON
PRINT "In main module. Press c to continue."

DO: LOOP UNTIL INKEY$ = "c"
```

```
CALL SubKey

PRINT "Back in main module. Press q to end."
DO : LOOP UNTIL INKEY$ = "q"
END

GotF1Key:
   PRINT "Handled F1 keystroke in main module."
RETURN

' ============================================================
'         SUBKEY MODULE
' ============================================================
SUB SubKey STATIC
   PRINT "In module with SUBKEY. Press r to return."

   ' Pressing F1 here still invokes the GotF1Key
   ' subroutine in the MAIN module:
   DO : LOOP UNTIL INKEY$ = "r"
END SUB
```

Output

```
In main module. Press c to continue.
Handled F1 keystroke in main module.
In module with SUBKEY. Press r to return.
Handled F1 keystroke in main module.
Back in main module. Press q to end.
Handled F1 keystroke in main module.
```

Compiling Programs From the Command Line

When compiling code containing any of these statements from the command line you must use one of the compiler options described in Table 8.1.

ON *event* **GOSUB**
event **ON**
EVENT ON

Table 9.1 Compiler Options for Event Trapping

Option	Action
/V	Causes BASIC to check between each program statement for the occurrence of an event.
/W	Causes BASIC to check only at each line number or line label for the occurrence of an event.

The /V option detects events sooner, however the program will run slower and take up more memory. As an alternative, you can add labels to your program at the places where you need detection and compile with the /W option as shown in this example:

```
ON KEY (1) GOSUB F1Handler
KEY (1) ON
.
.
.
' Check for an event here.
Checkpoint1:
' Continue processing without checking.
.
.
.
DO UNTIL Condition%
' Check for event every time we loop.
Checkpoint2:
.
.
.
LOOP
' No more event checking.
.
.
.
END
F1Handler:
' Code to take action when F1 is pressed goes here.
RETURN
```

Chapter 10
Database Programming with ISAM

Microsoft BASIC gives you the power and flexibility of Indexed Sequential Access Method (ISAM) through a group of straightforward statements and functions that are part of the BASIC language. ISAM statements and functions provide an efficient and simple method for quickly accessing specific records in large and complex data files. This chapter describes ISAM, its statements and functions, and how to use them in programs that access and manipulate the records in ISAM database files. These statements and functions make it easy for your programs to manage database files as large as 128 megabytes.

When you finish this chapter, you'll understand:

- What ISAM is, and when and why it is useful.
- The new and modified BASIC statements used for ISAM file access and manipulation.
- A general approach to creating, accessing, and manipulating records in ISAM databases.
- The structure of an ISAM file.
- Using indexes to work with data records as though they were sorted in many ways.
- Using EMS (expanded memory) with ISAM programs.
- Using transaction statements in applications with complex block processing requirements.
- Converting existing database code to ISAM code.
- Using ISAM utilities to convert your sequential and database files to ISAM format, to compact ISAM databases, repair damaged databases, and exchange tables between database files and sequential files.

Note ISAM is supported only in MS-DOS. You cannot use it in OS/2 programs.

What Is ISAM?

When a program uses or modifies records stored in a file, it often has to sort and re-sort the records in various ways. When a file contains many complex records, sorting can require substantial program code and a great deal of processing time. ISAM is an approach to creating and maintaining a special data file, in which the way records typically need to be sorted can be easily defined and efficiently stored along with the records themselves. This means your program doesn't have to re-sort the records each time the file is used or each time you want a different perspective on the records.

In addition to your data records, an ISAM file contains information that describes and facilitates access to each data record. Much of this information is maintained in "tables" and "indexes." Tables serve many purposes, including allowing quick access to any of the values of a specific data record. Indexes represent various ways of ordering the presentation of records in a table, and they permit you to easily access a whole record by the value of a field in the record.

ISAM statements and functions manipulate, present, and manage the records in ISAM data files. ISAM's record-searching and ordering algorithms are faster and more efficient than routines that you might create in BASIC to perform these tasks, so it not only saves you significant programming effort, but improves the speed and capacity of many database programs as well.

For database applications, ISAM files are more convenient and efficient than random-access files because they allow you to access the file as though the records were ordered in a variety of different ways. The next several sections compare ISAM to other types of files and introduce some concepts and terms that are helpful in understanding ISAM. (Traditional sequential and random-access files are discussed in Chapter 3, "File and Device I/O.")

ISAM Statements and Procedures

The statements for performing ISAM file tasks are integrated into the BASIC language. In most cases, new statements have been added for ISAM. In a few cases, existing statements have simply been expanded. The following list categorizes the ISAM statements by the type of task for which you use them:

Task	Statements
File and table creation/access	**OPEN, CLOSE, DELETETABLE, TYPE...END TYPE**
Controlling presentation order of data (indexing)	**CREATEINDEX, GETINDEX$, SETINDEX, DELETEINDEX**
Position change relative to the current record	**MOVEFIRST, MOVELAST, MOVENEXT, MOVEPREVIOUS, TEXTCOMP**
Position change by field value	**SEEKGT, SEEKGE, SEEKEQ**
Table information	**BOF, EOF, LOF, FILEATTR**
Data exchange	**INSERT, RETRIEVE, UPDATE, DELETE**
Transaction processing	**BEGINTRANS, COMMITTRANS, CHECKPOINT, ROLLBACK, SAVEPOINT**

Some ISAM statement usage rules parallel BASIC rules, while others are more specific due to the characteristics of the ISAM file. For example, the BASIC **LOF** function, which returns the length of a sequential file or the number of records in a random-access file, returns the number of records in the specified table when used on an ISAM file.

The **TYPE...END TYPE** statement is used to define the structure of the record variables that will be used to exchange data between your program and the ISAM file. The elements in a **TYPE...END TYPE** statement can have any user-defined type or BASIC data type except variable-length strings and dynamic arrays. However, the **TYPE...END TYPE** statement used for ISAM access cannot contain BASIC's **SINGLE** type. Floating-point numeric elements in a **TYPE...END TYPE** statement used for ISAM access must have **DOUBLE** type. Fixed-point decimal numeric elements can have BASIC's new **CURRENCY** type.

Similarly, when you name the elements of the user-defined type, you must name them according to the ISAM naming convention (which is a subset of the BASIC identifier-naming convention). The name of the user-defined type itself, however, is a BASIC identifier and follows the BASIC naming convention. Similarly, some arguments to ISAM statements follow BASIC naming conventions, while others follow the ISAM subset (described in the section "ISAM Naming Convention" later in this chapter).

Note The ISAM statements and functions are integrated into the BASIC language. However, within the QBX environment the ISAM statements are recognized, but cannot be executed unless you invoke a terminate-and-stay-resident (TSR) program before starting QBX. Using a TSR allows QBX to provide full ISAM support, but only when your programs need it. For programs that don't use ISAM, not loading (or unloading) the TSR saves substantial memory.

ISAM Vs. Other Types of File Access

ISAM files are often used in place of random-access files because ISAM provides more flexible access to any arbitrary record within the database. Although it is not an ASCII text file, an ISAM file is a sequential-access file. When an ISAM file is opened, BASIC uses ISAM routines that handle all interaction between the operating system and the actual file. ISAM organizes your data records into a structure called a table. You can think of this table as a series of horizontal "rows," each row corresponding to a full data record. The table is also divided into vertical "columns," each column corresponding to one of the fields in your data records.

With random-access files you use the **GET** statement to access a record by its record number (which represents the order in which the record was inserted into the file). Random access does not provide access to records based on the values in specific fields within the record. When you access an ISAM file, ISAM's indexing and **SEEK***operand* statements allow you to test the values in a specified group of "vertical" fields (columns) against a stated condition. Put another way, a random-access file is accessible by row number only, like a list. With an ISAM file you can access records either by relative position (row) or by the contents of any field in a specified column. The ISAM statements give you the ability to access specific records in the file with the speed of a random-access file, but with a great deal more flexibility.

The following list contrasts the unit of access used by BASIC file types:

File type	Access unit
Binary	By byte
Sequential	By line or by byte
Random	By record number (i.e., row only)
ISAM	By position, or by the value of any field (or group of fields) within a table

ISAM also differs from other record-indexing approaches because all the information relating to the records is contained in a single file that also contains the data records themselves. This can greatly facilitate user management of complicated databases without sacrificing speed and convenience.

The ISAM Programming Model

The following chart describes the general sequence of steps used in ISAM database programming. It compares the ISAM model and statements to the corresponding tasks and statements used with random-file access.

Task to be performed	ISAM approach	Random-file approach
Associate program variables with database records.	Use **TYPE...END TYPE**, and **DIM.**	Use **TYPE...END TYPE**, and **DIM.**
Access records.	Use **OPEN** to access a table.	Use **OPEN** to access a file.
Change presentation order of records according to value in a specified field.	Use **CREATEINDEX** and/or **SETINDEX.**	No support provided. Requires sorting code, unless record-insertion order is adequate.
Specify record to work with.	Use **MOVE***dest* to move by row or **SEEK***operand* to move to a record containing a specified field value.	Use **GET** to retrieve a record by record number, or if that is not adequate, requires searching code to determine which record to **GET.**

Data exchange.	Use **RETRIEVE** to assign a record from a table to program variables. Use **UPDATE** to assign program variables to a record in a table. Use **INSERT** to insert a record in a table. Use **DELETE** to delete a record from a table. When you overwrite, insert, or delete records, ISAM handles all table and index maintenance transparently.	Use **GET** and **PUT** for simple fetching and overwriting of existing records. To delete a record, you typically code to mark it for deletion; write a temporary file that omits it; then delete the original file; and finally, rename the temporary file with the original filename. To do a simple insert of a record, you must code to keep track of the number of records, then insert each new record as number n+1. Inserting at a specific position requires code to swap records.
Change presentation order of records to get a different perspective on data.	Use **CREATEINDEX** or **SETINDEX.**	Requires sorting code.
Close the file(s).	Use **CLOSE** for tables.	Use **CLOSE.**

ISAM Concepts and Terms

Many terms used in describing ISAM are familiar, however when used with ISAM many terms have specialized connotations. For example, when using random file access, one typically thinks of a file as a collection of logically related records. In ISAM such a collection of records is called a "table," since an ISAM disk file (called a "database") can contain multiple and distinct collections of records. This section explains some fundamental ISAM concepts and terms.

Field A single data item constituent of a record.

Record A collection of logically related data-item fields. The association of the fields is defined by a **TYPE...END TYPE** statement in your program.

Row Synonym for record. When placed in an ISAM table, the collection of fields in a specific record is referred to as a row. Thus, a row in a table corresponds to a single data record. See Figure 10.1.

Table An ordered collection of records (rows), each of which contains a single data record. The records in a table have some logical relationship to one another. The default order of records in the table corresponds to the order in which records were added.

Column Each column in a table has a name. A column in a table is the collection of all fields having the same column name. Thus, a column is a vertical collection of fields in the same way a row is a horizontal collection of fields. The name of each column in a table comes from the corresponding element in a **TYPE...END TYPE** statement in the program that created the

table. For example the **TYPE...END TYPE** statement in Figure 10.1 shows some of the elements of a record variable defined in a hypothetical program. When an ISAM table is created to accept data from records of `ExampleType`, ISAM uses the **TYPE...END TYPE** element names as the names of the table's columns. Figure 10.1 illustrates the relationships between rows, fields, and columns in a table.

```
TYPE ExampleType
      UnitsSent     AS  INTEGER
      TotalBill     AS  CURRENCY
      ClientCode    AS  DOUBLE
      .
      .
      .
      ClientRep     AS  STRING * 20
END TYPE
```

UnitsSent	TotalBill	ClientCode	. . .	ClientRep
UnitsSent $_1$	TotalBill $_1$	ClientCode $_1$	. . .	ClientRep $_1$
UnitsSent $_2$	TotalBill $_2$	ClientCode $_2$	. . .	ClientRep $_2$
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
UnitsSent $_n$	TotalBill $_n$	ClientCode $_n$	. . .	ClientRep $_n$

Column Names

Actual Rows in Table

Actual Columns in Table

Figure 10.1 Records and Fields (Rows and Columns)

Database A collection of tables and indexes contained in a disk file.

Index An independent structure within the ISAM file created when a **CREATEINDEX** statement is executed. Each index represents an alternative order for presentation of the records in the table. The order is based on the relative values of each data item in the column (or columns) specified in the **CREATEINDEX** statement. You might want to think of an index as a "virtual table," that is, a virtual ordering of the table's data records. An index must have a name, and may have other attributes. Any index you create is saved and maintained as part of the database until it is explicitly deleted. Figure 10.2 illustrates a table. Beside the table, a list of positions indicates where each record actually resides in the table. Beneath the table, a diagram illustrates how an index on one of the columns (the Invoice column) changes the presentation order of the data when the index is specified.

Default Presentation Order of Table

UnitsSent	ClientCode	Invoice	ClientRep	
105	147.914	6037.01	Smith & Jones	← *position 1*
172	162.694	5427.22	Duckworth Inc.	← *position 2*
166	155.555	2445.78	Charles R. Pierce	← *position 3*
197	149.276	6003.73	Mulholland Co.	← *position 4*
220	153.353	8072.47	Pinzer & Sons	← *position 5*
177	121.212	5002.00	Bonelli Behring	← *position 6*

Table Position

↓

```
CREATEINDEX filenum,"InvoiceIndex",0,"Invoice"
SETINDEX filenum,"InvoiceIndex"
```

↓

InvoiceIndex Entry	*Points to*	*The record at*
InvoiceIndex position 1	→	*table position 3*
InvoiceIndex position 2	→	*table position 6*
InvoiceIndex position 3	→	*table position 2*
InvoiceIndex position 4	→	*table position 4*
InvoiceIndex position 5	→	*table position 1*
InvoiceIndex position 6	→	*table position 5*

↓

New Presentation Order

UnitsSent	ClientCode	Invoice	ClientRep
166	155.555	2445.78	Charles R. Pierce
177	121.212	5002.00	Bonelli Behring
172	162.694	5427.22	Duckworth Inc.
197	149.276	6003.73	Mulholland Co.
105	147.914	6037.01	Smith & Jones
220	153.353	8072.47	Pinzer & Sons

Figure 10.2 Indexes on the Columns of a Table

Specifying an index is a separate step in which the order of the indexed column (or columns) is imposed on any presentation of the table's records. To present a table's records in the index's order, you first specify that index in a **SETINDEX** statement. If you were to create, and then

specify the index on the Invoice column in the preceding figure, the records would be presented in the order shown in the final part of Figure 10.2, with the record at table position 3 first, followed by the record at table position 6, followed by the record at table position 2, and so on.

NULL Index The default index for a table; that is, the presentation order of the records when no user-created index is specified. When the NULL index is in effect (for instance, when the table is first opened), the order of the records is the order in which they were inserted into the table. In Figure 10.2, this order is illustrated by the table itself.

Record Order The actual physical order of records on disk is arbitrary because whenever records are deleted from a table, their physical disk positions are filled by the next records inserted in the table. This optimizes access speed and disk-space usage. For example, if you delete the third record added to the table, then add the sixth record, the sixth record would be placed in the physical disk position previously occupied by the third record as shown in Figure 10.3.

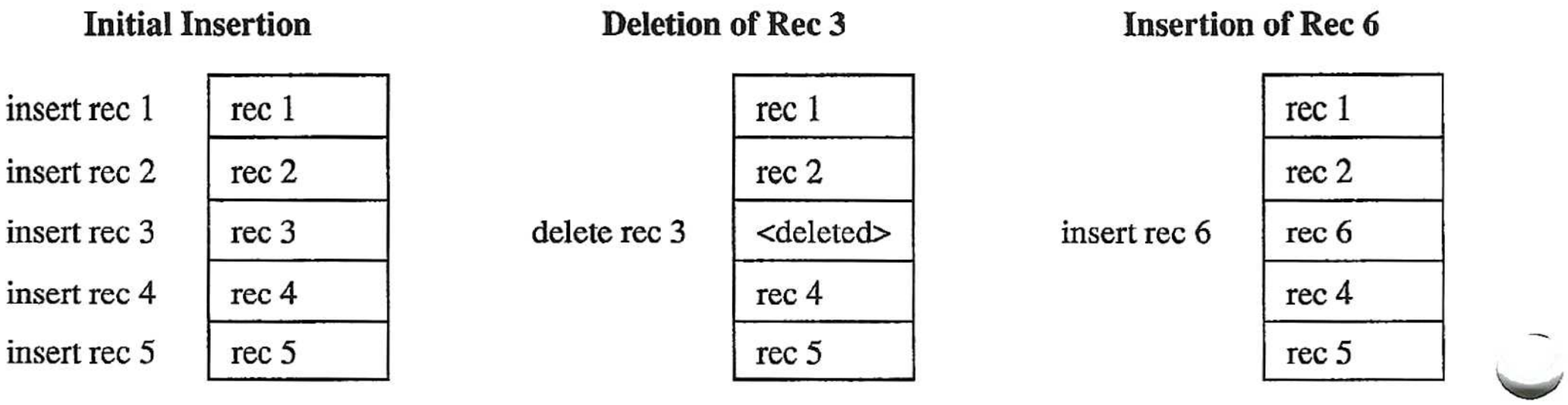


Figure 10.3 Insertion and Deletion of Records in a Table

Insertion Order The order in which the records are inserted into the table. This order corresponds to the order imposed by the NULL index (the default index).

Presentation Order The apparent order imposed on the table by the current index. Note that file-space optimization has the side effect that subordering of records when an index is applied corresponds to the actual physical order of records on the disk. If you want presentation order to include specific subordering, use a combined index.

Combined Index An index that is based on more than a single column. Specifying a combined index enforces a specific subordering on the presentation order of the records.

Indexed Value The value (or combination of values) that determines a record's position on a particular index. While an index is a collection of indexed values, there is one indexed value for each record (row). Therefore, with a combined index, the indexed value is the combination of the constituent fields of the index.

Key Value A value against which indexed values are tested when using a **SEEK***operand* statement to seek a record that meets the specified condition.

Unique Index An index requiring that each indexed value (i.e., each field in the column on which the index is defined) be different from all others. When you use the **CREATEINDEX** statement to create an index, you can specify it as "unique." For example, if you specified a

unique index on ClientCode column in the table in Figure 10.2, ISAM would generate a trappable error if the value of any ClientCode field ever duplicated the value of any other ClientCode field in the table. You could use a unique index to prevent a user from assigning the same ClientCode to two different clients.

Current Position The focus of activity in a table. Understanding "position" in the table is important because, in all cases, position is relative to some structure within the table that doesn't exist in other BASIC files. In an ISAM table, "currency" (meaning the current position, not the new **CURRENCY** data type) is determined by several factors. Since multiple indexes may be created on any table, the current position depends on which index has been specified. There is no concept of a "current table" in ISAM, but there is always a "current index" for each open ISAM table. If a table contains any records, one record is the "current record," except in two cases:

- There is no current record at the beginning or end of the table (when **BOF** or **EOF** return true).
- There is no current record after the unsuccessful execution of a **SEEK***operand* statement (because **EOF** then returns true).

Otherwise, every open table has a current index and a current record (if it contains any records).

Current Index The index whose sorting order determines the order of appearance of a table's records. When a table is first opened, the NULL index is the current index. Once an index is created on a column or combination of columns, specifying it as the current index imposes the sorting order of that column or combination of columns on the presentation order of all records in the table. That index remains the current index on that table until it is deleted, a different index is specified for that table, or the table is closed. Every open table has a current index.

Current Record The focus of activity between your program and an ISAM table. Directly following a **SETINDEX** statement, the current record is the record with the smallest indexed value, according to the index specified. ISAM provides many statements for making different records current, and for altering the current record. At any time, one and only one record can be the current record in each open table. When the NULL index is specified with **SETINDEX**, the current record is the record that was inserted into the table first.

Focus The focus of data exchange (that is, the record affected by an ISAM statement that fetches, overwrites, inserts, or deletes data in a table). The focus is always the current record, as determined by the current index.

ISAM Engine Routines used by ISAM to access and maintain the database.

ISAM File A database created and maintained using ISAM.

Data Dictionary Tables and indexes used by the ISAM engine in maintaining the database.

ISAM Components

ISAM works by applying routines contained in the ISAM engine to a physical disk structure called an ISAM file. The following sections describe these components and the table/index model for record access.

The ISAM Engine

The ISAM engine creates a table representing your data records to enhance the storage of and rapid access to each record. The relationship between the individual records in the table and what is actually in memory as your program runs is determined by the ISAM engine. This lets you easily manipulate the records of very large files as though they all fit in memory at once. The actual order of physical storage of records in the file is unimportant because ISAM allows you to deal with the records as though they were stored in a variety of convenient ways.

Your program can present the records in the file according to the sorting order of any column in the table simply by creating, and then specifying an index. Using the **SETINDEX** statement is functionally equivalent to sorting the records according to the values of the constituent fields of the indexed column (or combination of columns). When indexes are created, they become part of the database. They can then be saved as part of the database, or deleted. When saved, the ISAM file contains the indexes you've created, in addition to the tables containing the data records themselves. Indexes are described in detail in the section "Creating and Specifying Indexes on Table Columns" later in this chapter.

The Parts of the ISAM File

ISAM places your data records in the table (or tables) you specify. Each table represents a group of logically related records, but the logic of the groupings is completely up to you. For example, it might be useful to keep one table in the database for your inventory and another for clients.

Information describing an ISAM file is maintained within the file in a set of system tables called the data dictionary. The data dictionary itself is invisible to your program, and you never have to deal with it if you don't want to. In fact, it is safest to simply let the ISAM engine handle all interaction with the data dictionary, since corrupting it could destroy your database. Information in the data dictionary includes table names, indexes and index names, column names, and all the other information used by ISAM to access and manipulate the records in response to the ISAM statements.

You can create any number of tables within a database file, although the number of tables you can open simultaneously has an upper limit of 13 and decreases each time an additional database is opened. A practical maximum of four databases can be opened at once. The section "Using Multiple Files" describes how many tables can be opened, relative to the number of open databases.

ISAM File Allocation and Growth

Because an ISAM file contains descriptive information, it has some file-size overhead. Additionally, to optimize access speed and flexibility, ISAM files grow periodically in large chunks (32K per chunk), rather than in record-sized increments as single records are added. A database contains a header of about 3K. Each table has 4K of overhead beyond its actual data records; each index requires at least 2K. The data dictionary consists of five system tables plus eight system indexes, resulting in a total initial overhead of about 39K. Therefore, the smallest ISAM file is 64K. Though an ISAM file with a single record is 64K, there is considerable room for adding data records within that 64K file before the next 32K chunk is added. The initial combination of system tables and system indexes is about 39K; the remaining 25K are used for your data records and the new indexes and tables you create.

When to Use ISAM

For data files too large to completely load into memory, ISAM vastly simplifies file manipulation because ISAM support replaces the kind of sorting that can only otherwise be accomplished efficiently by loading all data records in memory simultaneously. This makes ISAM an excellent method for dealing with large amounts of data which require sorted access. An ISAM file can be as large as 128 megabytes. ISAM handles all the work of moving portions of such a huge file in and out of memory during record manipulation.

Whenever data records contain many fields that need to be examined in a variety of ways, using ISAM simplifies the programming. Although you can write code to sort or index random-access files, ISAM integrates these tasks for you with high-level statements that manipulate sophisticated file structures. This lets you easily manipulate records by the values in specific fields, and is far more flexible than a random file's one-dimensional ordering by record number. (For an example of how much BASIC code just one index for a random file requires, see the program INDEX.BAS listed in Chapter 3, "File and Device I/O.") However, if disk space is at a premium, don't automatically choose ISAM for short, easily-sorted files of relatively constant size. These may be better handled using other methods (for example, by creating and using hash tables). However, if you need to sort on different fields at different times, or if you need very fast access to records according to complex subsorting orders, the benefits of the ISAM file quickly make up for its overhead. Also, consider that for very large files, the amount of descriptive information relative to actual records remains relatively constant, so the percentage of the file devoted to overhead decreases progressively.

The Table/Index Model

In ISAM, tables and indexes represent various fundamental arrangements of the data records. When you insert a record in an ISAM table, its place in the table is the result of a process that optimizes file size and speed of access. References to the record are immediately placed in all of that table's existing indexes, including the NULL index, so the presentation order of all records is always internally consistent. The default order for a table is the chronological order of insertion.

Specifying an index other than the NULL index orders the records of the table by the sorting order of each field in the indexed column (or combination of columns). For example, if each row in a table contains five columns, you can create indexes on any, or all, or any combination of the five columns. When you specify one of these indexes (with **SETINDEX**), you impose the sort order of the index on the presentation order of the records. (The sort order is either numeric or alphabetic, depending on the data type of the column or columns). Specifying a different index changes the presentation order of the records. As you add and remove records from the table, ISAM maintains all relevant information about the table. Figure 10.4 illustrates a simple table in which each record is a collection of six user-defined fields. The table can be represented as four rows, each having six columns.

Number	Name	Phone	Birthday	Sex	Sport
1	Ted	505-2121	5/21	M	Golfing
2	Carol	555-3705	7/7	F	Golf
3	Alice	505-2121	9/3	F	Tennis
4	Bob	555-3705	6/18	M	Billiards

Figure 10.4 Table with Four Rows and Six Columns

The values in each field in the Number column represent the order in which each record was added to the table. In practice, you would probably never define such a column, since insertion order, the NULL index, is the default ordering of the records in an ISAM table, but it is included here for illustrative purposes. In a random-access file, the Number column would correspond to the record number, and would be the only way you could reference a record without writing special code to sort the file. However, because this is an ISAM table, you can use an index to specify a presentation order that corresponds to the sort order of any of the columns.

Suppose you wanted to organize a celebration, and you wanted to compile an invitation list that included only women. With just the **OPEN**, **CREATEINDEX**, and **SETINDEX** statements you could create and specify an index on the Sex column. Since F sorts before M, all the women in the table would be presented before any of the men, as shown in Figure 10.5.

Number	Name	Phone	Birthday	Sex	Sport
2	Carol	555-3705	7/7	F	Golf
3	Alice	505-2121	9/3	F	Tennis
1	Ted	505-2121	5/21	M	Golfing
4	Bob	555-3705	6/18	M	Billiards

Figure 10.5 Presentation Order of Table Sorted by Sex Column

With this order the program could easily start from the first record, display its data, then use the **MOVENEXT** statement to make each successive record the current record. Conversely, if you wanted to invite only men, the program could start from the last record (using the **MOVELAST** statement), and then use **MOVEPREVIOUS** to traverse the records in reverse order. As the program displayed each previous record, you could choose among the men.

An index can be specified on each column of a table. Figure 10.6 presents the table information indexed on the Sport column.

Number	Name	Phone	Birthday	Sex	Sport
4	Bob	555-3705	6/18	M	Billiards
2	Carol	555-3705	7/7	F	Golf
1	Ted	505-2121	9/21	M	Golfing
3	Alice	505-2121	5/3	F	Tennis

Figure 10.6 Presentation Order of Table Sorted by Sports Column

You can also create combined indexes by "combining" the values in several fields so records appear sorted, first by one field, then sorted by another, and so on. For example, you could create and specify a combined index that presented the records sorted first by the Phone column, then by the Birthday column. This would present the records sorted first by household, then within each household, by the order of the birthdays of each person with the same phone number, as shown in Figure 10.7.

Number	Name	Phone	Birthday	Sex	Sport
3	Alice	505-2121	5/3	F	Tennis
1	Ted	505-2121	9/21	M	Golfing
4	Bob	555-3705	6/18	M	Billiards
2	Carol	555-3705	7/7	F	Golf

Figure 10.7 Presentation Order of Table Sorted First by Phone Column, then by Birthday Column

The preceding examples are for illustrative purposes. Normally, when designing a program you would provide the user with several useful indexes on the records, rather than designing the program to let the user create indexes as needed. However, if you want to let users create their own indexes, you can do so using the ISAM statements and functions.

A Sample Database

The following sections describe a table within an ISAM database file that could be used by a library to keep track of its inventory of books. Examples demonstrate how to create or open the database and view the records from several different perspectives. (This program, BOOKLOOK.BAS, as well as its associated .MAK file (BOOKLOOK.MAK), secondary modules (BOOKMOD1.BAS, BOOKMOD2.BAS, BOOKMOD3.BAS), database file (BOOKS.MDB), and include file (BOOKLOOK.BI) are included on the disks supplied with Microsoft BASIC. When you ran the Setup program, they were placed in the directory you specified for BASIC source and include files.

Designing the BookStock Table

Inventory maintenance of a book-lending library can be used to illustrate the ISAM approach. Assume that the patrons of the library are concerned only with books dealing with the BASIC programming language. For example, you can create a database containing a single table that includes pertinent information about all the library's books about BASIC. Figure 10.8 illustrates the form such a table might take.

```
TYPE Books
      IDNum       AS  DOUBLE
      Price       AS  CURRENCY
      Edition     AS  INTEGER
      Title       AS  STRING * 50
      Publisher   AS  STRING * 50
      Author      AS  STRING * 36
END TYPE
```

Book Stock

IDnum	Price	Edition	Title	Publisher	Author
15561.1276	22.95	1	QuickBASIC ToolBox	Microsoft Press	Craig, J.C.
15561.5125	19.95	2	QuickBASIC V. 4.0	Microsoft Press	Herger, D.
.					
.					
.					
47161.3011	35.00	1	Using MS QuickBASIC	Wiley	Cooper, J.

Figure 10.8 The BookStock Table in the BOOKS.MDB Database

Creating, Opening, and Closing a Table

The statements used for creating, opening, and closing databases and tables are the familiar BASIC **TYPE...END TYPE**, **OPEN**, and **CLOSE** statements. However, they are used differently with ISAM files than with other types of files.

Naming the Columns of the Table

The first step in creating a table is to include an appropriate **TYPE...END TYPE** statement in the declarations part of your program. The name you use for this user-defined type is an argument to the **OPEN** statement that creates the database file and the table, and may be used subsequently whenever the table is opened. When the table is first created, the names used for the elements in the **TYPE...END TYPE** statement become the names of the corresponding columns in the table (See Figure 10.8).

Specifying the Data Types of the Columns

What can appear in a column of a table is determined by the column's data type. For instance, a column having **INTEGER** type can accept whole numbers in the normal integer range. Similarly, a column having **STRING** type can contain a string as large as 32K. Assigning data types to the columns makes it possible for ISAM to create the indexes that can be used to change the presentation order of the table's records.

Note

Although you can fetch and write data that has the following characteristics to an ISAM table, you cannot create indexes on them:

- **STRING** columns longer than 255 bytes.
- Columns with aggregate (i.e., array) type.
- Columns with structure (i.e., user-defined) type.

Each column in a table has the data type specified in the **TYPE...END TYPE** statement used as the *tabletype* in the **OPEN** statement that created the table. Data types you specify in an ISAM **TYPE...END TYPE** statement must be one of those shown in Table 10.1.

Table 10.1 Valid BASIC Data Types Within ISAM Files

Data type	Size limit	Description	Indexable
INTEGER	2 bytes, signed	Whole numbers in the range –32,768 to 32,767.	Yes
LONG	4 bytes, signed	Whole numbers in the range –2,147,483,648 to 2,147,483,647.	Yes

Table 10.1 *Continued*

Data type	Size limit	Description	Indexable
DOUBLE	8 bytes	Real numbers in the following ranges: –1.7976931348623l5D 308 to –4.94065D –324, zero, and 4.94065D–324 to 1.7976931348623l5D308 .	Yes
CURRENCY	8 bytes	Use to store fixed-point (such as dollar and cents) values in the approximate range ±9.22E14 with accuracy to 19 digits.	Yes
STRING	Up to 32K	Use to store string data.	Only if shorter than 256 bytes
Static Array	64K	Raw binary	No
User-defined type (structure)	64K	Raw binary	No

Note that BASIC's **SINGLE** data type is not legal in ISAM; use **DOUBLE** or **CURRENCY** instead. The following declaration can be used in a program that creates or accesses the sample BookStock table.

```
TYPE Books
   IDnum     AS DOUBLE         ' ID number for this copy
   Price     AS CURRENCY       ' Original cost of book
   Edition   AS INTEGER        ' Edition number of book
   Title     AS STRING * 50    ' The title of the book
   Publisher AS STRING * 50    ' The Publisher's name
   Author    AS STRING * 36    ' The author's name
END TYPE
```

Although BASIC would accept element identifiers up to 40 characters long, the elements in this statement must follow the ISAM naming convention (see the section "ISAM Naming Convention" later in this chapter), since they will become the names of columns within the ISAM database file. The actual name of the user-defined type can be any valid BASIC identifier however, because it is never actually used within the ISAM file.

Data Type Coercion

Although BASIC performs considerable data type coercion in other situations, the only coercion performed between your BASIC program and ISAM is in relation to **SEEK***operand* statements, and even then only between **INTEGER** and **LONG** values. Therefore, if a **LONG** value is expected by ISAM, and you pass an **INTEGER**, the **INTEGER** will be coerced to a

LONG, and no type-mismatch error is generated. However, if you try to pass a **LONG** when ISAM expects an **INTEGER**, coercion may result in an `Overflow` error. In other situations, such as passing a **CURRENCY** value when a **DOUBLE** is expected, a `Type mismatch` error is generated. Since BASIC's default data type is **SINGLE** precision numeric, passing a literal (even 0) to ISAM can cause a type mismatch (since **SINGLE** is not a valid ISAM data type). In such a case, you should append the type-declaration character for the type expected by ISAM to the number. Even if you reset the default data type with a **DEF***type* statement, it is a good idea to screen the types of all numbers passed to ISAM to make sure they are properly typed and that they will fit within the range of the expected type.

Opening the BookStock Table

The declaration of the user-defined type is all the preparation a simple program needs to prepare for opening an ISAM database and table. The following code can now be used to create or open the table within the ISAM file:

```
' You could write code here to check to see if the file exists,
' then open the file if it does or display a message if it doesn't.

OPEN "BOOKS.MDB" FOR ISAM Books "BookStock" AS # 1
```

Using OPEN and CLOSE with ISAM

To open a table, you use the traditional BASIC **OPEN** statement with arguments and clauses specific to ISAM. Whenever you open a table, you must specify the database file that contains the table. The syntax for an ISAM **OPEN** is as follows:

OPEN *database$* **FOR ISAM** *tabletype tablename$* **AS** [#]*filenumber%*

Argument	Description
database$	A string expression representing a DOS filename, so it follows operating-system file-naming restrictions. This argument can include a drive letter and a path.
tabletype	A BASIC identifier that specifies a user-defined type already declared in the program. Note that, unlike the other arguments, it cannot be a string expression.
tablename$	A string expression that follows the ISAM naming convention.
filenumber%	An integer within the range 1–255, the same as in the traditional BASIC **OPEN** statement. Note that *filenumber%* is associated with both the *tablename* of the table being opened and the database file (*database$*) itself containing the table. Therefore, the same *database$* can appear in any number of **OPEN** statements, each of which opens a different table (with a unique *filenumber%*) in the same database file. You can use **FREEFILE** to get available values for *filenumber%*.

String arguments are *not* case sensitive, so you can use inconsistent capitalization in any of these references.

The **CLOSE** statement is the same for an ISAM database as for any other file:

CLOSE [[#]] [[*filenumber%*]] [[, [[#]]*filenumber%*]]...

Opening a Table

The **FOR ISAM** clause simply replaces the **FOR OUTPUT** (or **APPEND**, or **INPUT**) clause used for other sequential-file access. The ISAM engine then handles all file interaction.

The behavior of an **OPEN...FOR ISAM** statement is similar to **OPEN...FOR OUTPUT** or **OPEN...FOR APPEND** with other types of sequential files. For example, if *database$* does not yet exist as a disk file, it is created by the **OPEN** statement. Similarly, if *tablename* does not exist within the database, the **OPEN** statement creates a table of that name within the database, and opens it. The *tabletype* argument must identify a user-defined type previously declared in the program with a **TYPE...END TYPE** statement. This precludes writing programs that permit the end user to design custom tables at run time. If an ISAM **OPEN** statement fails, all ISAM buffers are written to disk and any pending transactions are committed (See the section "Block Processing with Transactions" later in this chapter for information on transactions.)

Note

You cannot lock an ISAM database using **OPEN...FOR ISAM.** However, you can open a database that has been designated read-only by some other process. If your program opens such a file, certain ISAM statements will generate errors, including: **DELETE**, **DELETEINDEX**, **DELETETABLE**, **CREATEINDEX**, **INSERT**, and **UPDATE.** These statements cause `Permission denied` error messages.

Closing a Table

A **CLOSE** statement with *filenumber%* as an argument closes the *tablename$* associated with *filenumber%*. **CLOSE** with no arguments closes all open tables (and any other files, ISAM or otherwise). Any ISAM **CLOSE** statement causes all pending transactions to be committed. (See the section "Block Processing with Transactions" later in this chapter for information on transactions.)

The Attributes of filenumber%

A program that opens an ISAM table can open other files for other types of access. In such cases, you may need to determine at some point which files (or tables), associated with which file numbers, are open for which types of access. **FILEATTR** has the following syntax:

FILEATTR(*filenumber%*, *attribute%*)

When you use this function, you pass the number of the file or table you want to know about as the first argument, and either 1 or 2 as *attribute%*. If you pass a 1, the value returned in **FILEATTR** is 64 if *filenumber%* was opened as an ISAM table.

Other return values indicate the file was opened for another mode, as follows:

FILEATTR return value	Mode
1	**INPUT**
2	**OUTPUT**
4	**RANDOM**
16	**APPEND**
32	**BINARY**
64	**ISAM**

With an ISAM file, if you pass a 2 as *attribute%*, **FILEATTR** returns zero.

Defining a Record Variable

Although it isn't necessary to do it at the same time you open the database, you eventually need to define a record variable having the proper user-defined type for the table. This variable is used in transferring data between your program and the ISAM file. In the case of the `Books` type, it could look like this:

```
DIM Inventory AS Books
```

Creating and Specifying Indexes on Table Columns

Much of the power of ISAM derives from the ease with which the apparent order of data records can be changed. This is accomplished by specifying a previously created "index" on a column (or columns) in a **SETINDEX** statement.

If you don't specify an index on a table, the default index (the NULL index) is used, and the apparent order of the records is the order in which the records were added to the file. Therefore, when you initially open a table, the current index is the NULL index until (and unless) you specify a different index. You create your own indexes with the **CREATEINDEX** statement, using the following syntax:

CREATEINDEX [[#]]*filenumber%, indexname$, unique%, columnname$* [[, *columnname$*]]...

Argument	Description
filenumber%	The integer used to open the table on which the index is to be created.
indexname$	A string expression that follows the ISAM naming conventions. The index is known by *indexname$* until explicitly deleted.
unique%	A numeric expression. A non-zero value specifies a unique index on the column, meaning that no values in any of that column's fields can duplicate any of the others. A value of zero for this argument means the indexed values need not be unique.

columnname$* — A string expression following the ISAM naming convention that specifies the column to be indexed. Note that multiple *columnname$ entries do not create multiple independent indexes, but rather create a single combined index. Each succeeding *columnname$* identifies a subordinate sorting order for the records (when that index is specified).

The **CREATEINDEX** statement is used only once for each index. If you try to create an index that already exists for the table, a trappable error is generated.

Once an index exists, you use the **SETINDEX** statement to make it the current index (thereby imposing its order on the presentation order of the records). **SETINDEX** has the following syntax:

SETINDEX [[#]]*filenumber%* [[,*indexname$*]]

An *indexname$* argument is mandatory in the **CREATEINDEX** statement, but optional with **SETINDEX**. If you do not specify an index name with **SETINDEX**, the NULL index becomes the current index. Immediately after execution of **SETINDEX**, the specified index is the current index, and the current record becomes the record having the lowest sorting value in that column.

Note

Comparisons made by ISAM when sorting strings differ somewhat from those performed by BASIC. When collating, the case of characters is not significant, and trailing blank spaces are stripped from a string before comparison is made. In strings that are otherwise identical, accents are significant, and collating is performed based on the choice you made when running the Setup program. English, French, German, Portuguese, and Italian comprise the default group. Dutch and Spanish each have their own collating orders, and the Scandinavian languages (Danish, Norwegian, Finnish, Icelandic, and Swedish) comprise the fourth group. See Appendix E, "International Character Sort Order Tables," in the *BASIC Language Reference* for specifics of each group.

Indexes on BookStock's Columns

For example, assuming the BookStock table illustrated in Figure 10.8 was opened as # 1, you could use the **CREATEINDEX** statement to create the index `TitleIndexBS` (on the table's Title column) as follows:

```
CREATEINDEX 1, "TitleIndexBS", 0, "Title"
```

After this definition, you can use **SETINDEX** to make this index the current index as follows:

```
SETINDEX 1, "TitleIndexBS"
```

Once specified as current, the index represents a virtual table in which the data records are ordered according to the values in the Title column. This index imposes an alphabetic presentation order on the table. Therefore, all copies of books entitled *QuickBASIC Made Easy* would appear in sequence, and precede copies of *QuickBASIC ToolBox*, and so forth.

Creating a Unique Index

In the BookStock table, the IDnum column contains numbers for each copy of each book in the library. When new copies of a book are acquired, each is given a unique number. In this example all copies of *QuickBASIC ToolBox* have a whole number part of 15561, but each has a different fractional part. The following statement creates an index on this column and passes a non-zero value as the *unique%* argument, ensuring that no duplicate `IDnum` values can be entered for any copies of any books:

```
CREATEINDEX 1, "IDIndex", 1, "IDnum"
```

If there are already duplicates in the column, a trappable error is generated when your program attempts to execute the **CREATEINDEX** statement. If your program ever attempts to update the table with a duplicate value in a field in a unique index, a trappable error is generated.

In determining whether string values are duplicates, comparisons are case-insensitive and trailing blanks are ignored. Accented letters are not duplicates of their unaccented counterparts.

Subordering of Records Within an Indexed Column

There are many cases in which a user might need or expect a specific subordering. For example, when browsing a group of customer orders, if you are traversing an index based on account numbers, you might expect the actual orders associated with a specific customer name to be in the order in which the records were added. To ensure that records are presented in the way your user expects, you can create a combined index.

In the BookStock table example, the idea of multiple records representing multiple copies of a specific book in the library means that the `TitleIndexBS` index created previously would be suitable for a librarian who wanted to see quickly how many copies of a specific book the library owned. The librarian also might want to have the books presented in the order in which they were purchased. Creating, then specifying a combined index on the Title and Author columns would present all of the books with a specific title/author combination grouped together. When you index on a column that contains the same value in more than one record, the subordering of records with the same value for the column (in this case, the combination of Title and Author columns) is unpredictable because many table entries may have been inserted and deleted at different times. The books would appear in the order in which their records actually appear on the disk. ISAM optimizes for space by allowing new records to be placed on the disk in space previously occupied by deleted records. Therefore, although the title/author combined index would group the presentation of all copies of books titled *QuickBASIC ToolBox* and written by D. Hergert, it would present them in the order in which they appear on the disk.

However, the `IDnum` for each copy would correspond to the dates when each was added to the collection (assuming the library did not reuse an old `IDnum` once an old copy of a book was replaced). A combined index that included the Title, Author, and IDnum columns would present the records with the oldest copy appearing first among that group of titles by that author, and so on.

Similarly, if the library had purchased three hard-bound and five paperback copies, the difference would show up as a significant differential in price. In presenting these books in the database, a librarian might want to have all the hard-bound copies appear in sequence, separated from the paperback copies. A combined index on the Title, Author, and Price columns would create that presentation order (assuming paperbacks of a specific title are always cheaper than their hardbound counterparts). If the IDnum column were added to create a Title, Author, Price, IDnum index, then all the paperbacks would appear in the order in which they were added to the collection before any of the hard-bound copies.

Creating a Combined Index

A combined index can be created using as many as 10 columns in a table by listing multiple *columnname$* arguments in the **CREATEINDEX** statement. When such a multi-column index is the current index, the records appear as though first sorted by the first *columnname$*. Then those records whose first indexed values are identical appear as though sorted by the next *columnname$*, and so on. The following example creates a combined index:

```
CREATEINDEX 1, "BigIndex", 0, "Title", "Author", "IDnum"
```

When **SETINDEX** is used to specify `BigIndex` as the current index, the records appear sorted first by `Title`. The same title might appear in the database for several books by different authors, but making the `Author` the second part of the combined index would keep all those by a particular author grouped. Finally, giving the `IDnum` as the last part of the index would cause the oldest copy of the desired book (by the given author) to be presented first in its group.

You can designate a combined index as unique. If you do so, only the combination must be unique. For example, a unique index on the Author and Title columns would permit any number of occurrences of the same author *or* the same title, but only for one instance of the same author and the same title.

Null Characters Within Indexed Strings in a Combined Index

If you place null characters within strings in columns that are components of a combined index, there are situations in which the order of the index may deviate from what you expect. This is rare, but results from the fact that ISAM uses the null character as a separator in combined indexes. For instance, if the last character in a string field is a null, and its index is combined with one whose first character is a null, and in all other ways they are the same, the two fields will compare equal. This applies only to combined indexes. There is no restriction on null characters in an ISAM string, but you should be aware of this situation if you plan to use null characters in strings of indexed columns.

Practical Considerations with Indexes

Remember that each time an **INSERT**, **DELETE**, or **UPDATE** statement is executed, every index in the affected table is adjusted to reflect the changed state of the records. In the normal course of moving through a database and making changes to records, the time needed to adjust indexes would not be noticeable to a user. However, the time required by an automated process that makes these types of changes may be significantly affected by the number of indexes in the table. In some cases it may make sense to delete unnecessary indexes before such a process begins, then recreate them when it is finished.

Restrictions on Indexing

Part of the information ISAM maintains is the data type of each column in each table. These data types are stored in the data dictionary so the ISAM engine can make valid comparisons when sorting records in an index. ISAM can index columns having up to 255 bytes in combined length. Therefore, you can create an index on a string column having a length of up to 255 bytes, or a combined index whose constituent columns total a little less than 255 bytes or less (there is a little overhead associated with each constituent index). Columns having array or structure (that is, user-defined) type cannot be indexed.

Attempting to create an index on a column with array or structure type, or on a **STRING** column longer than 255 characters, or defining a combined index whose total length is greater than 255 bytes, causes a trappable error.

Determining the Current Index

The **GETINDEX$** function lets you find out what the current index is. **GETINDEX$** has the following syntax:

GETINDEX$ (*filenumber%*)

The *filenumber%* argument is an integer identifying any open table. **GETINDEX$** returns a string representing the name of the current index. If the value returned in **GETINDEX$** is a null string (represented by ""), then the current index is the NULL index.

In a complex program, it may become difficult to predict which index is the current index on a specific table. Although **SETINDEX** is a single call, it is usually more efficient to test the current index with **GETINDEX$** first, then only use **SETINDEX** if you actually want a different index. The following fragment illustrates this:

```
IF GETINDEX$(TableNum%) <> "MyIndex" THEN
   SETINDEX TableNum%, "MyIndex"
END IF
```

Even though the preceding is more code than simply using **SETINDEX**, it is usually more efficient. Note also that the effect on the current record may be different depending on whether the **SETINDEX** statement is executed. If the current index is already MyIndex, the current record will be the same (on that index) as it was previously. If the **SETINDEX** statement is executed, the current record will be the one that sorts lowest in the index.

Transferring and Deleting Record Data

The syntax for the data-manipulation statements is similar to that for **SETINDEX**:

DELETE [[#]] *filenumber%*

RETRIEVE [[#]] *filenumber%, recordvariable*

UPDATE [[#]] *filenumber%, recordvariable*

INSERT [[#]] *filenumber%, recordvariable*

Argument	Description
filenumber%	The integer used to open the table whose current record you want to remove, fetch, or overwrite. In the case of an insertion, it is the table in which you want the record inserted.
recordvariable	A variable of the user-defined type corresponding to the table into which the current record values are placed, or with which the current record is to be overwritten. In the case of an insertion, *recordvariable* is the record you wish to insert. Its elements (in its **TYPE...END TYPE** declaration) may be exactly the same as those of the table, or a subset of them. Subsets of *recordvariable* are discussed in the section "Record Variables as Subsets of a Table's Columns" later in this chapter.

RETRIEVE, UPDATE, and **DELETE** all refer to the current record. The data transfer statements all take the data in *recordvariable* and either place it in the table (**UPDATE** and **INSERT**) or fetch the current record (**RETRIEVE**) and place its data into *recordvariable.*

DELETE removes the current record from the specified table, and all affected indexes are adjusted appropriately. Following a deletion, if the current record was not the last record in the current index, the new current record is the record that immediately succeeded the deleted record. If the deleted record was the last record, no record is current, and **EOF** returns true.

When you use **RETRIEVE**, the contents of the current record are assigned to *recordvariable*.

When you use **UPDATE**, the contents of *recordvariable* overwrite the current record, and all affected indexes are adjusted appropriately.

A trappable error occurs if no record is current when a **DELETE, RETRIEVE,** or **UPDATE** statement is executed.

INSERT places the contents of *recordvariable* in the table, then adjusts all affected indexes appropriately. A newly inserted record assumes its appropriate position in the current index. Therefore, if you display the current record immediately after an insertion, the record displayed is the same record that was displayed prior to the insertion, not the newly inserted record. To see the new record, execute a **SETINDEX** statement to make the NULL index current, then execute a **MOVELAST** statement, then display the current record. The **INSERT** statement

itself does not affect positioning. A trappable error occurs if you try to insert a record containing a duplicate value in a column on which a unique index exists.

The Current Position

The current position within a table depends on that table's current index. When you specify an index with **SETINDEX**, you specify the table (with the *filenumber%* argument) and the index name. After **SETINDEX** is executed, the current record is the first record on the specified index. The current record is the focus of data exchange.

Changing the Current Index

After opening a table, and until you specify an index, the current index is the NULL index. To change to another index, use the **SETINDEX** statement. It has the following syntax:

SETINDEX [[#]] *filenumber%*, [[, *indexname$*]]

Argument	Description
#	The optional number character.
filenumber%	The integer used to open the table for which you want to set a new current index.
indexname$	A string expression naming a previously created index. If *indexname* is omitted, the NULL index becomes the current index, otherwise *indexname* becomes the current index.

Making a Different Record Current

ISAM permits you to make records current either by their position within the current index (using a **MOVE***dest* statement), or by testing field value(s) in the current index against key value(s) you supply (in a **SEEK***operand* statement).

Setting the Current Record By Position

The **MOVE***dest* statements let you make a record in the specified table current based on its position in the current index. Use the **BOF** and **EOF** functions to test the current position in the table. When a move is made, it is relative to the current position on the table specified by the *filenumber%* argument.

The syntax for the **MOVE***dest* statements and the position-testing functions is as follows:

MOVEFIRST [[#]] *filenumber%*

MOVELAST [[#]] *filenumber%*

MOVENEXT [[#]] *filenumber%*

MOVEPREVIOUS [[#]] *filenumber%*

EOF (*filenumber%*)

BOF (*filenumber%*)

Each record in a table has a previous record and a next record, except the records that are first and last according to the current index. Given the current index, the beginning of the table is the position *preceding* the first record; the end of the table is the position *following* the last record.

The effect of any of the **MOVE***dest* statements, or position-testing functions, is relative to the current position in the table specified by *filenumber%*. If there is a record following the current record, **MOVENEXT** makes it the current record. If there is a record preceding the current record, **MOVEPREVIOUS** makes it the current record. An attempt to use **MOVENEXT** from the last record in the table, or to use **MOVEPREVIOUS** from the first record in the table moves the position to the end of file, or the beginning of file, respectively.

You can test for the end-of-file and beginning-of-file conditions with the **EOF** and **BOF** functions. **EOF** returns true (–1) when the current position is beyond the last record on the current index; **BOF** returns true (–1) when the current position precedes the first record on the current index.

If the current record is not already the first or last in the table, then **MOVEFIRST** and **MOVELAST** make those records current. When a table contains no records, both **BOF** and **EOF** return true (–1). If the table has no records, an attempt to execute any of the **MOVE***dest* statements will fail and **EOF** will return true.

Displaying the BookStock Table

When your program specifies an index for the first time after opening a table, the current record is the one that sorts first in the index. Some preliminary BASIC code, plus the ISAM **RETRIEVE**, **BOF**, **EOF**, **MOVENEXT**, and **MOVEPREVIOUS** statements, are all you need to allow the user to move through an open table and view its records.

A Typical ISAM Program

Although you can write your program to allow users to create their own database files, it is more typical to supply the program with a database file having no records, with the filename hard-coded into the program. This database would contain all the predefined indexes you anticipate your user would need. Once the file contains these indexes, your program doesn't

need the code used to create them. Supplying an empty database file that contains all the necessary indexes simplifies user interaction with the program, and also means ISAM can use less memory. For more information on the different ways you can include ISAM support in your programs, and how to use it during program development, see the sections "Starting ISAM for Use in QBX" and "Using ISAM with Compiled Programs" later in this chapter.

The following example shows the module-level code of a program that opens several tables, including the BookStock table (discussed earlier) in the BOOKS.MDB database. It allows the user to view the records for all the books in a variety of orders, depending on which index is chosen. Only the module-level code and the `Retriever` procedure appear here. Other procedures, most of which control the user interface (to let the user add, edit, and search for specific records), are called as necessary. You can see those procedures in the disk files listed in the .MAK file BOOKLOOK.MAK (including BOOKLOOK.BAS, the main module; BOOKMOD1.BAS; BOOKMOD2.BAS; and BOOKMOD3.BAS). As noted previously, these example files are included on the Microsoft BASIC distribution disks and may be copied to your hard disk during Setup.

Note that the error-handling routine at the bottom of the code handles error 86, `Illegal operation on a unique index`. When an attempt is made to update or insert a record containing a duplicate value for a unique index, the error-handling routine prompts the user to enter a new value.

To view the BOOKLOOK.BAS sample application, move to the directory where it was installed during Setup and invoke the ISAM TSR and QBX by typing the following two lines:

```
PROISAM /Ib:24
QBX  BOOKLOOK
```

The /Ib argument to PROISAM specifies the number of buffers ISAM will need to manipulate data. Options for the ISAM TSR are fully explained in the section "Starting ISAM for Use in QBX" later in this chapter. You can see the effect of a combined index by using the Title+Author+ID index on the BookStock table in the example. The library contains five copies of the title *Structured BASIC Applied to Technology* by Thomas A. Adamson. Using the combined index, the various copies of the book are presented in ID number order, whether you are moving forward in the table or backwards. If you just use the Title index (or any of the other indexes for which the fields are duplicates), the order moving forward is probably what you would expect, but the order moving backward may surprise you. This illustrates the fact that specific subordering is only guaranteed when a specific combined index is current.

The `Retriever` procedure illustrates fetching a record from the database and placing it in a defined *recordvariable*. `CheckPosition` updates the Viewing/Editing keys box when the first or last record in the table is reached. The following listing begins with selected lines from the include file BOOKLOOK.BI.

```
' Some definitions and declarations contained in BOOKLOOK.BI

' Define constants for TRUE and FALSE
CONST FALSE = 0, TRUE = NOT FALSE

' Define constants for the database file & table names
```

```
CONST cBookStockTableNum = 1, cCardHoldersTableNum = 2
CONST cBooksOutTableNum = 3, cDisplayedTables = 2

' Define names similar to keyboard names with their equivalent key codes.
CONST SPACE = 32, ESC = 27, ENTER = 13, TABKEY = 9, ESCAPE = 27
CONST BACKSPACE = 8, DOWN = 80, UP = 72, LEFT = 75, RIGHT = 77
CONST HOME = 71, ENDK = 79, PGDN = 81, PGUP = 73

' Screen positions - Initialized for 25 rows. Screen positions can be
' modified for 43-row mode if you have an EGA or VGA adapter, in which case
' two tables could probably be displayed at the same time.
CONST TABLETOP = 1, TABLEEND = 14, BOXTOP = 16, NLINE = 17, RLINE = 18
CONST WLINE = 19, VLINE = 20, ALINE = 21, ELINE = 22, CLINE = 23
CONST BOXEND = 24, INDBOX = 42, HELPCOL = 1, SCREENWIDTH = 74
CONST MESBOXTOP = TABLEEND, MESFIELD = TABLEEND + 1
CONST MESBOXEND = BOXTOP
CONST DATATOP = 2, DATAEND = 13, NAMEFIELD = 3, TITLEFIELD = 3
CONST STREETFIELD = 5, AUTHORFIELD = 5, CITYFIELD = 7, PUBFIELD = 7
CONST STATEFIELD = 9, EDFIELD = 9, ZIPFIELD = 11, PRICEFIELD = 11
CONST CARDNUMFIELD = 13, IDFIELD = 13

' Assign constants to the operations the user can request. These are all
' dealt with in the main DO loop of the module-level code.
CONST QUIT = 0, GOAHEAD = 1, GOBACK = 2, BORROWER = 3, WHICHBOOKS = 4
CONST SEEKFIELD = 5, REORDER = 6, STATUS = 7, OTHERTABLE = 8, TOSSRECORD = 9
CONST ADDRECORD = 10, CHECKOUT = 11, CHECKIN = 12, INVALIDKEY  = 13
EDITRECORD = 14, CONST UNDO = 15,UNDOALL = 16
CONST BIGINDEX = 20, NULLINDEX = 21      ' Used in OrderCursor & PlaceCursor
CONST KEYSMESSAGE = "Press a Viewing/Editing key", EMPTYSTRING = ""

' Create a structure type for the BookStock table.
TYPE Books
  IDnum     AS DOUBLE
  Price     AS CURRENCY
  Edition   AS INTEGER
  Title     AS STRING * 50
  Publisher AS STRING * 50
  Author    AS STRING * 36
END TYPE

' Create a structure type for the CardHolders table.
TYPE Borrowers
  CardNum          AS LONG
  Zip              AS LONG
  TheName          AS STRING * 36
  City             AS STRING * 26
```

```
  Street          AS STRING * 50
  State           AS STRING * 2
END TYPE
' Create a structure type for the BooksOut table.
TYPE BookStatus
  IDnum     AS DOUBLE
  CardNum   AS LONG
  DueDate   AS DOUBLE
END TYPE
' Create a structure type handling all the tables in the database.
TYPE RecStruct                      ' This structure contains each of the other
  TableNum    AS INTEGER            ' other table structures. When you pass a
  WhichIndex  AS STRING * 40        ' reference to this structure to procedures
  Inventory   AS Books              ' the procedure decodes the TableNum
  Lendee      AS Borrowers          ' element, then deals with the proper table.
  OutBooks    AS BookStatus         ' WhichIndex is used to communicate the index
END TYPE                            ' the user wants to set.


'******************************** Main  Module ********************************
'*                                                                            *
'*      The main module of BOOKLOOK contains two DO loops. The outer loop     *
'*      permits cycling among the database's displayable tables. The inner    *
'*      loop accepts user input and calls procedures to carry out tasks.      *
'*                                                                            *
'******************************************************************************
DEFINT A-Z
'$INCLUDE: 'BOOKLOOK.BI'
SCREEN 0
CLS                             ' TempRec is used for editing and adding records
DIM TempRec AS RecStruct        ' Used only to blank out a TempRec
DIM EmptyRec AS RecStruct       ' See BOOKLOOK.BI for declaration of
DIM BigRec AS RecStruct         ' this structure and its elements
' Open the database and the BookStock, CardHolders, and BooksOut tables

ON ERROR GOTO MainHandler
OPEN "BOOKS.MDB" FOR ISAM Books "BookStock" AS cBookStockTableNum
OPEN "BOOKS.MDB" FOR ISAM Borrowers "CardHolders" AS cCardHoldersTableNum
OPEN "BOOKS.MDB" FOR ISAM BookStatus "BooksOut" AS cBooksOutTableNum
ON ERROR GOTO 0

BigRec.TableNum = cBookStockTableNum    ' Decide which table to show first

   ' Since the database has multiple tables, this outer DO loop is used to
   ' reset the number associated with the table the user wants to
   ' to access, then draw the screen appropriate to that table, etc.
```

```
   ' Only the Retriever procedure is shown in this listing.
DO
  EraseMessage                              ' See disk file for procedures
  CALL DrawScreen(BigRec.TableNum)          ' not shown in this listing.
  Checked = CheckIndex%(BigRec, TRUE)       ' Show current index
  IF (EOF(BigRec.TableNum) AND BOF(BigRec.TableNum)) THEN
    CALL ShowMessage("There are no records in this table", 0): SLEEP
  ELSE
    CALL Retriever(BigRec, DimN, DimP, Answer) ' Retrieve and show a record.
    CALL ShowMessage(" Press V to View other table", 0)
    CALL ShowStatus(" Total records in table: ", CDBL(LOF(BigRec.TableNum)))
  END IF

  ' This loop lets the user traverse BigRec.TableNum and insert, delete,
  ' or modify records. It also permits presentation of the records according
  ' to the indexes created by the CREATEINDEX statements in the MakeOver
  ' procedure. Indexes support record searching according to a specific
  ' value (called an "indexed value") in the current index.
  DO                                            ' At start of each loop, show
                                                ' the user valid operations
    CALL Retriever(BigRec, DimN, DimP, Answer) ' and display current record.

    STACK 4000                                  ' Set large stack for recursions-it
                                                ' also resets FRE(-2) to stack 4000.

    Answer% = GetInput%(BigRec)                 ' Find out what the user wants to do.

    IF Answer < UNDO THEN                       ' Excludes UNDOALL too.
      CALL EditCheck(PendingFlag, Answer, BigRec)
    END IF

    SELECT CASE Answer                  ' Process valid user requests
      CASE QUIT
      CALL ShowMessage(" You chose Quit. So long! ", 0)
      END

                                        ' If user picks "N" (Next Record), MOVENEXT.
                                        ' CheckPosition handles end-of-file (i.e. the
  CASE GOAHEAD, ENDK                    ' position just past the last record). If EOF
                                        ' or BOF = TRUE, CheckPosition holds position.
    MOVENEXT BigRec.TableNum
    CALL CheckPosition(BigRec, Answer, DimN, DimP)

                                        ' Same logic as GOAHEAD, but reversed
  CASE GOBACK, HOME
```

```
    MOVEPREVIOUS BigRec.TableNum
    CALL CheckPosition(BigRec, Answer, DimN, DimP)

                                ' If user chooses "E", let him edit a field.
                                ' Assign the value returned by SAVEPOINT to
                                ' an array element, then update the table and
                                ' show the changed field. Trap any "duplicate
  CASE EDITRECORD               ' value for unique index" (error 86) and
                                ' handle it. The value returned by SAVEPOINT
                                ' allows rollbacks so the user can undo edits.

    IF EditField(Argument%, BigRec, Letter$, EDITRECORD, Answer%) THEN

        ' You save a sequence of save point identifiers in an array so
        ' you can let the user roll the state of the file back to a
        ' specific point. The returns from SAVEPOINT aren't guaranteed
        ' to be sequential.
      n = n + 1                 ' Increment counter first so savepoint
      Marker(n) = SAVEPOINT     ' is sycnced with array-element subscript.

      Alert$ = "Setting Savepoint number " + STR$(Marker(n))
      CALL ShowMessage(Alert$, 0)
      ON ERROR GOTO MainHandler
      SELECT CASE BigRec.TableNum   ' Update the table being displayed.
        CASE cBookStockTableNum
          UPDATE BigRec.TableNum, BigRec.Inventory
        CASE cCardHoldersTableNum
          UPDATE BigRec.TableNum, BigRec.Lendee
      END SELECT
      ON ERROR GOTO 0
  ELSE
      COMMITTRANS               ' Use COMMITTRANS abort transaction if
      PendingFlag = FALSE       ' the user presses ESC.
      n = 0                     ' Reset array counter.
   END IF
                              ' If choice is "A", get the values the user wants
                              ' in each of the fields (with AddOne). If there
                              ' is no ESCAPE from the edit, INSERT the record.
                              ' Trap "Duplicate value for unique index" errors
                              ' and handle them in MainHandler (error 86).
  CASE ADDRECORD
    added = AddOne(BigRec, EmptyRec, TempRec, Answer%)
    IF added THEN
      Alert$ = "A new record assumes proper place in current index"
      CALL ShowMessage(Alert$, 0)
      ON ERROR GOTO MainHandler
```

```
    SELECT CASE BigRec.TableNum      ' Insert into table being shown
      CASE cBookStockTableNum
      INSERT BigRec.TableNum, TempRec.Inventory
      CASE cCardHoldersTableNum
      INSERT BigRec.TableNum, TempRec.Lendee
    END SELECT
    ON ERROR GOTO 0
  END IF
  TempRec = EmptyRec

                          ' If choice is "D" --- prompt for confirmation.
                          ' If so, delete it and show new current record.
CASE TOSSRECORD
  Alert$ = "Press D again to Delete this record, ESC to escape"
  CALL ShowMessage(Alert$, 0)
  DeleteIt% = GetInput%(BigRec)
  IF DeleteIt% = TOSSRECORD THEN    ' Delete currently-displayed record.
    DELETE BigRec.TableNum
    CALL ShowMessage("Record deleted...Press a key to continue", 0)
  ELSE
    CALL ShowMessage("Record not deleted. Press a key to continue", 0)
    CALL ShowRecord(BigRec)
  END IF
  ' The following code checks whether the record deleted was the last
  ' record in the index, then makes the new last record current
  IF EOF(BigRec.TableNum) THEN
    MOVELAST BigRec.TableNum
  END IF

                          ' If user chooses "R", walk the fields so he
                          ' can choose new index to order presentation.
CASE REORDER
  Letter$ = CHR$(TABKEY)
  GotOne = ChooseOrder(BigRec, EmptyRec, TempRec, Letter$, REORDER)

                          ' If a choice of indexes was made, retrieve
                          ' the index name, set an error trap, and try
                          ' to set the index, then display new index.
IF GotOne THEN
  IndexName$ = LTRIM$(RTRIM$(TempRec.WhichIndex))
  ON ERROR GOTO MainHandler
  IF IndexName$ <> "NULL" THEN              ' This string is placed in
    SETINDEX BigRec.TableNum, IndexName$    ' TempRec.WhichIndex if
  ELSE                                      ' user chooses "Default."
    SETINDEX BigRec.TableNum, ""            ' "" is valid index name
  END IF                                    'representing NULL index
```

```
      ON ERROR GOTO 0                              '(i.e. the default order).
      CALL AdjustIndex(BigRec)
      LSET TempRec = EmptyRec
    END IF
                        ' If choice is "F", first set current index
  CASE SEEKFIELD        ' using same procedure as REORDER. Then do seek.

    Letter$ = CHR$(TABKEY)          ' Pass TABKEY for PlaceCursor.
    GotOne = ChooseOrder(BigRec, EmptyRec, TempRec, Letter$, SEEKFIELD)

    IF GotOne THEN
      CALL SeekRecord(BigRec, TempRec, Letter$)
      FirstLetter$ = ""
    END IF

      CASE INVALIDKEY                  ' Alert user if wrong key is pressed.
        CALL ShowMessage(KEYSMESSAGE$, 0)
    END SELECT
    CALL DrawHelpKeys(BigRec.TableNum)
    CALL ShowKeys(BigRec, BRIGHT + FOREGROUND, DimN, DimP)
  LOOP
LOOP
CLOSE
END

' This error-handling routine takes care of the most common ISAM errors.
MainHandler:
IF ERR = 5 THEN                  ' 5 = Illegal function call
  CALL ShowMessage("The Illegal function call error was trapped", 0)
  RESUME NEXT

ELSEIF ERR = 73 THEN             ' 73 = Feature unavailable
  CALL ShowMessage("You forgot to load the ISAM TSR program", 0)
  END
ELSEIF ERR = 88 THEN             ' 88 = Database inconsistent
  ' If you have text files corresponding to each of the tables,
  ' MakeOver prompts for their names and creates an ISAM file from them.
  ' If you use a name other than BOOKS.MDB for the ISAM file name, you
  ' have to change the OPEN statements in BOOKLOOK.BAS to the new name.
  CALL MakeOver(BigRec)
  RESUME NEXT

ELSEIF ERR = 83 THEN             ' 83 = Index not found
    CALL DrawScreen(BigRec.TableNum)
    CALL ShowMessage("Unable to set the index. Need more buffers?", 0)
    RESUME NEXT
```

```
ELSEIF ERR = 86 THEN          ' 86 = Duplicate value for unique index
    ' If a user tries to enter a value for the Card Number or ID fields
    ' that duplicates a value already in the table, this trap prompts
    ' for a different value, then resumes execution. This error can also
    ' occur if the MakeOver procedure tries to insert duplicate values
    ' from a text file.
    IF BigRec.TableNum = cBookStockTableNum THEN
      DO
        Alert$ = STR$(BigRec.Inventory.IDnum) + " is not unique. "
        CALL ShowMessage(Alert$, 1)
        COLOR YELLOW + BRIGHT, BACKGROUND
        INPUT "Try another number: ", TempString$
        BigRec.Inventory.IDnum = VAL(TempString$)
      LOOP UNTIL BigRec.Inventory.IDnum
    ELSEIF BigRec.TableNum = cCardHoldersTableNum THEN
      DO
        Alert$ = STR$(BigRec.Lendee.CardNum) + " is not unique. "
        CALL ShowMessage(Alert$, 1)
        COLOR YELLOW + BRIGHT, BACKGROUND
        INPUT "Try another number: ", TempString$
        BigRec.Lendee.CardNum = VAL(TempString$)
      LOOP UNTIL BigRec.Lendee.CardNum
    END IF
    COLOR FOREGROUND, BACKGROUND
    RESUME
ELSE
  Alert$ = "Sorry, not able to handle this error in BOOKLOOK: " + STR$(ERR)
  CALL ShowMessage(Alert$, 0)
  END
END IF

DEFINT A-Z
'******************************** Retriever SUB ********************************
'* The Retriever SUB retrieves records from the database file and puts     *
'* them into the appropriate recordvariable for the table being displayed. *
'* An error trap is set in case the retrieve fails, in which case a message*
'* is displayed. Note that if a preceding SEEKoperand fails, EOF is TRUE.  *
'* In that case, position is set to the last record, which is retrieved.   *
'*                                                                         *
'*                              Parameters:                                *
'*   Big Rec       User-defined type containing all table information      *
'*   DimN & DimP   Flags telling which menu items should be dimmed/changed *
'*   Task          Tells what operation retrieve results from             *
'*******************************************************************************
SUB Retriever (BigRec AS RecStruct, DimN, DimP, Task)
  STATIC PeekFlag          ' Set this if user is just peeking at other table.
```

```
LOCATE , , 0                    ' Turn off the cursor.
' Show the user which choice was made, and whether EOF or BOF
CALL ShowKeys(BigRec, FOREGROUND + BRIGHT, DimN, DimP)
IF Task < EDITRECORD THEN                                    ' Edit needs its
  CALL Indexbox(BigRec, CheckIndex%(BigRec, 0))              ' own prompts. Show
END IF                                                       ' indexbox otherwise.
ON LOCAL ERROR GOTO LocalHandler                  ' Trap errors on the retrieve.
IF NOT EOF(BigRec.TableNum) THEN                  ' Retrieve current record
  SELECT CASE BigRec.TableNum                     ' from table being displayed
    CASE cBookStockTableNum                       ' if EOF is not true.
      RETRIEVE BigRec.TableNum, BigRec.Inventory
    CASE cCardHoldersTableNum
      RETRIEVE BigRec.TableNum, BigRec.Lendee
  END SELECT
ELSE                                              ' If EOF is true, set position
  MOVELAST BigRec.TableNum                        ' to the last record in table,
  SELECT CASE BigRec.TableNum                     ' then retrieve the record.
    CASE cBookStockTableNum
    RETRIEVE BigRec.TableNum, BigRec.Inventory
    CASE cCardHoldersTableNum
    RETRIEVE BigRec.TableNum, BigRec.Lendee
  END SELECT
  DimN = TRUE
END IF

ON LOCAL ERROR GOTO 0                                        ' Turn off error trap.
CALL ClearEm(BigRec.TableNum, 1, 1, 1, 1, 1, 1)
CALL ShowRecord(BigRec)
IF Task = OTHERTABLE THEN    ' If user is just peeking at the other table
  IF PeekFlag = 0 THEN       ' remind him/her how to get back to first table.
    CALL ShowMessage("Press V to return to the other table", 0)
    PeekFlag = 1
  END IF
  ELSE
    PeekFlag = 0
  END IF
EXIT SUB

LocalHandler:
  IF ERR = 85 THEN
    CALL ShowMessage("Unable to retrieve your record...", 0)
  END IF
  RESUME NEXT
END SUB
```

The `MakeOver` procedure referred to in the error-handling routine is not included in the listing, but illustrates how indexes can be created. You can use `MakeOver` (and the `Reader` procedure that it calls) to create an ISAM database containing the same tables as BOOKS.MDB, but containing records read from text files. The text files must be in the appropriate directory, but they don't all have to contain records. For example, you wouldn't necessarily want to start a database with entries in the BooksOut table. If the database already exists, the new records are appended to the appropriate table. In that case, the `Duplicate definition` error is generated when an attempt is made to create the indexes. The error is trapped, and the procedure ends. The fields of the text files should be comma delimited, and strings should be enclosed in double quotation marks.

Setting the Current Record by Condition

You can specify conditions to be met when making a record current with the **SEEKGT**, **SEEKGE**, and **SEEKEQ** statements. Their syntax is summarized as follows:

SEEK*operand filenumber%* [[, *keyvalue* [[, *keyvalue*]]...]]

Depending on the *operand* and the current index, these statements make the first matching record in the table specified by *filenumber* the current record. A match occurs when an indexed value fulfills the *operand* condition with respect to the specified *keyvalue.*

The following table indicates the operation associated with each of the *operand* specifiers:

Statement	Makes this record current
SEEKGT	The first record whose indexed value is greater than *keyvalue*.
SEEKGE	The first record whose indexed value is greater than, or equal to, *keyvalue*.
SEEKEQ	The first record whose indexed value equals *keyvalue*.

The *keyvalue* expression should have the same data type as the column represented by the current index. Although type coercion is performed between **INTEGER** and **LONG** values, you can experience overflow errors if you rely on automatic type coercion. With all other types, a *keyvalue* error is detected when a value is of the wrong data type, and a `Type mismatch` error results. If the current index is a combined index, and the number of *keyvalue* values exceeds the number of constituent columns in the combined index, a `Syntax error` error message is generated when the program runs. A trappable error is generated if you attempt to execute a **SEEK***operand* statement while the NULL index is the current index.

If the number of *keyvalue* values is fewer than the number of constituent columns in a combined index, the missing values are replaced by a value less than the smallest possible value, and the outcome depends on the *operand*. For example, a **SEEKEQ** will fail (and **EOF** will return true). A **SEEKGE** or **SEEKGT** will perform the seek based on whatever *keyvalue* values are supplied, and will find the first record that matches the supplied *keyvalue* values. For example, assume the following type, variable, and index are created:

```
TYPE ForExample
   FirstName     AS STRING * 20
   LastName      AS STRING * 25
END TYPE
DIM  ExampleVariable AS ForExample
CREATEINDEX TableNum%, "FullNameIndex", 0, FirstName, LastName
SETINDEX TableNum%, "FullNameIndex"
```

If you execute the following statement, the seek will fail because no last name was provided:

```
SEEKEQ TableNum%, "Tom"
```

If you execute the following statement, the seek will find the first record for which FirstName is "Tom":

```
SEEKGT TableNum%, "Tom"
```

When a **SEEK***operand* statement fails (that is, when no match is made), there is no current record, and **EOF** is set to true. Therefore, any immediately succeeding operation that depends on the current record (such as a **RETRIEVE**, **DELETE**, **INSERT**, **MOVENEXT**, or **MOVEPREVIOUS**) will cause a trappable error. You can prevent generation of this error by only executing statements that depend on the existence of a current record if **EOF** returns false.

Seeking on Strings and ISAM String Comparison

When seeking a string, ISAM performs comparisons on a case-insensitive basis. Trailing blanks are ignored. This is a less strict comparison than that made by the BASIC equality operator. Additionally, international conventions are observed (see Appendix E, "International Character Sort Order Tables," in the *BASIC Language Reference* for more information). The **TEXTCOMP** function allows you to perform string comparisons within your program in the same way they are compared by ISAM. Its syntax is as follows:

TEXTCOMP (*string1$*, *string2$*)

The *string1$* and *string2$* arguments are string expressions. **TEXTCOMP** returns –1 if *string1$* compares less than *string2$*, 1 if *string1$* compares greater than *string2$*, and 0 if the two strings compare equal. Only the first 255 characters of the respective strings are compared. For instance, if `BigRec` is the name of the *recordvariable* into which **RETRIEVE** places records from the BookStock table, and you want to print a quick list of the titles in the table that begin with the word QuickBASIC, you could use code like the following to find the first qualified title, then print the ensuing qualified titles:

```
IF GETINDEX$ (cBookStockTableNum%) <> "TitleIndexBS" THEN
   SETINDEX cBookStockTableNum%, "TitleIndexBS"    ' Set the index
END IF                                             ' if necessary.
SEEKGE cBookStockTableNum%, "quickbasic"           ' Find first one.
DO
   IF EOF(cBookStockTableNum%) THEN END            ' Quit at end of table.
   RETRIEVE cBookStockTableNum%, BicRec.Inventory ' Fetch record.
```

```
    ' Compare retrieved record to "quickbasic" the same way comparisons are
    ' done by ISAM. If they don't compare equal, exit from the loop.
    IF TEXTCOMP(LEFT$(BigRec.Inventory.Title, 10), "quickbasic") <> 0 THEN
       EXIT DO
    END IF
    LPRINT BigRec.Inventory.Title          ' Print qualified title.
    MOVENEXT BookStockTableNum%            ' Move on.
LOOP
```

Because the comparison performed by **TEXTCOMP** is case-insensitive, all variations of titles whose first word is QuickBASIC will be printed.

Example The following listing begins with the fragment of the module-level code of BOOKLOOK.BAS that handles the `SEEKFIELD` case, which is selected when the user chooses Find Record. First the user is prompted to choose an index by `ChooseOrder`, the same procedure called in the `REORDER` case. Then, the `SeekRecord` procedure is called. It prompts the user to enter a value to search for on the chosen index. After the user enters the value, he or she is prompted to choose the condition (=, >, >=, <, or <=) that controls the search. The default search is set in this example to use **SEEKEQ**, although the **SEEKGE** statement would be a better default in many cases.

`SeekRecord` calls procedures including `ValuesAccepted` (to make sure the input values have the same format as values in the tables), `ClearEm` and `ShowIt` (to show users what will be sought), `GetKeyVals` (in case the user is supplying values for a combined index), and `GetOperand` (to let the user choose whether the seek will be based on equality, greater or less than, greater than or equal to, or less than or equal to). Note that `ShowIt` shows what the user has entered in its appropriate field, not data from the database file itself. Other procedures include `MakeString`, `DrawScreen`, `ShowRecord` (which shows a database record), `ShowMessage`, `EraseMessage`, and `IndexBox`, all of which keep the user interface updated with each operation. Since names are in a single field, with last name first, `TransposeName` checks the format of a name entered and puts it in the proper format for searching or displaying.

```
' Module-level code dealing with Seeking values in fields...
   .
   .
   .
                        ' If choice is "F", first set current index
  CASE SEEKFIELD        ' using same procedure as REORDER. Then do seek.

    Letter$ = CHR$(TABKEY)          ' Pass TABKEY for PlaceCursor
    GotOne = ChooseOrder(BigRec, EmptyRec, TempRec, Letter$, SEEKFIELD)

    IF GotOne THEN
      CALL SeekRecord(BigRec, TempRec, Letter$)
      FirstLetter$ = ""
    END IF
```

```
        .
        .
        .

DEFINT A-Z
'******************************** ChooseOrder FUNCTION ************************
'*                                                                            *
'* The ChooseOrder FUNCTION calls PlaceCursor so the user can move around    *
'* the form to pick the index to set. PlaceCursor places the highlight in    *
'* successive fields and highlights the corresponding Index when the Task    *
'* parameter = SEEKFIELD or REORDER. When user presses enter in PlaceCursor* *
'* the name of the Index is placed in the WhichIndex element of RecStruct.   *
'*                                                                            *
'*                                  Parameters:                               *
'*  BigRec        BigRec has all the table information in updated form        *
'*  EmptyRec      EmptyRec is same template as BigRec, but fields are empty   *
'*  TempRec       Holds intermediate and temporary data                       *
'*  FirstLetter   Catches letter if user starts typing during SEEKFIELD       *
'*  Task          Either REORDER or SEEKFIELD - passed on to PlaceCursor      *
'*                                                                            *
'******************************************************************************
FUNCTION ChooseOrder (BigRec AS RecStruct, EmptyRec AS RecStruct,_
                      TempRec AS RecStruct, FirstLetter$, Task%)
  CALL DrawTable(BigRec.TableNum)
  CALL DrawIndexBox(BigRec.TableNum, Task)
  Argument = TITLEFIELD                          ' Always start with first field.
  TempRec = EmptyRec: TempRec.TableNum = BigRec.TableNum

  ' Pass temporary RecStruct variable so user can't trash BigRec.
  Value = PlaceCursor(Argument, TempRec, FirstLetter$, 1, Task)

  ' If the user chooses ESC, redraw everything, then exit to module level.
  IF ASC(TempRec.WhichIndex) = 0 THEN
    CALL DrawIndexBox(BigRec.TableNum, Task)
    CALL ShowRecord(BigRec)
    CALL ShowMessage(KEYSMESSAGE$, 0)
    ChooseOrder = 0
    EXIT FUNCTION
  ELSE                                           ' Otherwise, if user makes a choice
    ChooseOrder = TRUE                           ' of Indexes, signal success to the
  END IF                                         ' module-level code.
END FUNCTION
```

```
DEFINT A-Z
'********************************* SeekRecord SUB ****************************
'*  SeekRecord takes the name of the user's chosen index, sets it as the  *
'*  current index, then prompts the user to enter the value to seek. A    *
'*  minimal editor, MakeString, gets user input. If the SEEK is on a com- *
'*  bined index, GetKeyVals is called to get the input. Input is checked  *
'*  for minimal acceptability by ValuesOK. If it is OK, GetOperand is     *
'*  called to let the user specify how to conduct the SEEK.               *
'*                                                                        *
'*                              Parameters:                               *
'*      TablesRec  Contains current record information for all tables     *
'*                                                                        *
'*      TempRec    Contains the name of the index on which to seek (in    *
'*                 TempRec.WhichIndex element)                            *
'*                                                                        *
'*       Letter$   If the user starts typing instead of pressing ENTER    *
'*                 Letter$ catches the keystroke, passes it to MakeString *
'**************************************************************************
SUB SeekRecord (TablesRec AS RecStruct, TempRec AS RecStruct, Letter$)
  DIM EmptyRec AS RecStruct                    ' Make an empty record.
  IF LEFT$(Letter$, 1) < " " THEN              ' Exit if value is not a valid
                                               ' character, then redraw.
    CALL Indexbox(TablesRec, CheckIndex%(TablesRec, FALSE))
    CALL ShowMessage("You must enter a valid string or numeric value", 0)
    EXIT SUB
  END IF
  TheTable = TablesRec.TableNum
  IndexName$ = RTRIM$(TempRec.WhichIndex)
  IF GETINDEX$(TheTable) <> IndexName$ THEN    ' If index to seek on is not
    ON LOCAL ERROR GOTO SeekHandler            ' current, set it now. Trap
    SETINDEX TheTable, IndexName$              ' possible failure of SETINDEX
    ON LOCAL ERROR GOTO 0                      ' then turn off error trap.
  END IF
  CALL AdjustIndex(TablesRec)                  ' Show the current index.
  TablesRec.WhichIndex = TempRec.WhichIndex
  TempRec = EmptyRec                           ' Clear TempRec for data.
  TempRec.TableNum = TablesRec.TableNum

  ' Get the value to SEEK for from the user. The data type you assign the
  ' input to must be the same as the data in the database, so get it as a
  ' string with MakeString, then convert it to proper type for index. If
  ' the index is the combined index BigIndex, use GetKeyVals for input.
SELECT CASE RTRIM$(LTRIM$(IndexName$))
  CASE "TitleIndexBS", "AuthorIndexBS", "PubIndexBS", "NameIndexCH", "StateIndexCH"
    Prompt$ = "Value To Seek: "
    Key1$ = MakeString$(ASC(Letter$), Prompt$): IF Key1$ = "" THEN EXIT SUB
```

```
CASE "IDIndex", "CardNumIndexCH", "ZipIndexCH"
 ValueToSeek$ = MakeString$(ASC(Letter$), Prompt$)
 IF ValueToSeek$ = "" THEN EXIT SUB
 IF IndexName$ = "IDIndex" THEN
   NumberToSeek# = VAL(ValueToSeek$)
   Key1$ = ValueToSeek$
 ELSE
   NumberToSeek& = VAL(ValueToSeek$)
   Key1$ = ValueToSeek$
 END IF
CASE "BigIndex"
 CALL GetKeyVals(TempRec, Key1$, Key2$, Key3#, Letter$)
 ValueToSeek$ = STR$(Key3#)
 CASE ""
   Alert$ = "Sorry, can't search for field values on the default index"
   CALL ShowMessage(Alert$, 0)
 CASE ELSE
END SELECT

' Make sure the input values are minimally acceptable.

IF NOT ValuesOK(TablesRec, Key1$, Key2$, ValueToSeek$) THEN
  CALL ShowMessage("Sorry, problem with your entry. Try again!", 0)
  EXIT SUB
END IF

' Show the user the values he entered in their appropriate fields.
CALL ClearEm(TablesRec.TableNum, 1, 1, 1, 1, 1, 1)
CALL ShowIt(TempRec, IndexName$, TheTable, Key1$)

' GetOperand lets user specify the way the SEEK is to be conducted ---
' either  =, >, >=, <, or <= the value that was entered above.
DidIt = GetOperand%(Operand$)

' The actual SEEK has to be done according to two factors, the Index on
' which it is conducted, and the condition chosen in GetOperand. In the
' next section, use SELECT CASE on Operand$, then IF and ELSEIF on the
' basis of the index on which the search is being conducted.
 IF Operand$ <> "<>" THEN                    ' "<>" represents user ESC choice.

 SELECT CASE Operand$
  CASE "", "="                               ' If operand ="" or "=", use =.
    IF IndexName$ = "BigIndex" THEN
      IF INSTR(Key2$, ",") = 0 THEN Key2$ = TransposeName(Key2$) ' a name
      SEEKEQ TheTable, Key1$, Key2$, Key3#
```

```
    ELSEIF IndexName$ = "NameIndexCH" OR IndexName$ = "AuthorIndexBS" THEN
      IF INSTR(Key1$, ",") = 0 THEN Key1$ = TransposeName(Key1$) ' a name
      SEEKEQ TheTable, LTRIM$(RTRIM$(Key1$))
    ELSEIF IndexName$ = "IDIndex" THEN
      SEEKEQ TheTable, NumberToSeek#
    ELSEIF IndexName$ = "CardNumIndexCH" OR IndexName$ = "ZipIndexCH" THEN
      SEEKEQ TheTable, NumberToSeek&
    ELSE
      SEEKEQ TheTable, Key1$
    END IF
  CASE ">="                          ' at least gets them close
    IF IndexName$ = "BigIndex" THEN
      IF INSTR(Key2$, ",") = 0 THEN Key2$ = TransposeName(Key2$) ' a name
      SEEKGE TheTable, Key1$, Key2$, Key3#
    ELSEIF IndexName$ = "NameIndexCH" OR IndexName$ = "AuthorIndexBS" THEN
      IF INSTR(Key1$, ",") = 0 THEN Key1$ = TransposeName(Key1$)
      SEEKGE TheTable, Key1$
    ELSEIF IndexName$ = "IDIndex" THEN
      SEEKGE TheTable, NumberToSeek#
    ELSEIF IndexName$ = "CardNumIndexCH" OR IndexName$ = "ZipIndexCH" THEN
      SEEKGE TheTable, NumberToSeek&
    ELSE
      SEEKGE TheTable, Key1$
    END IF
  CASE ">"
    IF IndexName$ = "BigIndex" THEN
      IF INSTR(Key2$, ",") = 0 THEN Key2$ = TransposeName(Key2$)
      SEEKGT TheTable, Key1$, Key2$, Key3#
    ELSEIF IndexName$ = "NameIndexCH" OR IndexName$ = "AuthorIndexBS" THEN
      IF INSTR(Key1$, ",") = 0 THEN Key1$ = TransposeName(Key1$)
      SEEKGT TheTable, Key1$
    ELSEIF IndexName$ = "IDIndex" THEN
      SEEKGT TheTable, NumberToSeek#
    ELSEIF IndexName$ = "CardNumIndexCH" OR IndexName$ = "ZipIndexCH" THEN
      SEEKGT TheTable, NumberToSeek&
    ELSE
      SEEKGT TheTable, Key1$
    END IF
  CASE "<="
    IF IndexName$ = "BigIndex" THEN
      IF INSTR(Key2$, ",") = 0 THEN Key2$ = TransposeName(Key2$)
      SEEKGT TheTable, Key1$, Key2$, Key3#
      MOVEPREVIOUS TheTable
    ELSEIF IndexName$ = "NameIndexCH" OR IndexName$ = "AuthorIndexBS" THEN
      IF INSTR(Key1$, ",") = 0 THEN Key1$ = TransposeName(Key1$)
      SEEKGT TheTable, Key1$
```

```
      MOVEPREVIOUS TheTable
    ELSEIF IndexName$ = "IDIndex" THEN
      SEEKGT TheTable, NumberToSeek#
      MOVEPREVIOUS TheTable
    ELSEIF IndexName$ = "CardNumIndexCH" OR IndexName$ = "ZipIndexCH" THEN
      SEEKGT TheTable, NumberToSeek&
      MOVEPREVIOUS TheTable
    ELSE
      SEEKGT TheTable, Key1$
      MOVEPREVIOUS TheTable
    END IF
  CASE "<"
    IF IndexName$ = "BigIndex" THEN
      IF INSTR(Key2$, ",") = 0 THEN Key2$ = TransposeName(Key2$)
      SEEKGE TheTable, Key1$, Key2$, Key3#
      MOVEPREVIOUS TheTable
    ELSEIF IndexName$ = "NameIndexCH" OR IndexName$ = "AuthorIndexBS" THEN
      IF INSTR(Key1$, ",") = 0 THEN Key1$ = TransposeName(Key1$)
      SEEKGE TheTable, Key1$
      MOVEPREVIOUS TheTable
    ELSEIF IndexName$ = "IDIndex" THEN
      SEEKGE TheTable, NumberToSeek#
      MOVEPREVIOUS TheTable
    ELSEIF IndexName$ = "CardNumIndexCH" OR IndexName$ = "ZipIndexCH" THEN
      SEEKGE TheTable, NumberToSeek&
      MOVEPREVIOUS TheTable
    ELSE
      SEEKGE TheTable, Key1$
      MOVEPREVIOUS TheTable
    END IF
  CASE ELSE
    Alert$ = "The returned operand was " + Operand$
    CALL ShowMessage(Alert$, 0)
       SLEEP
   END SELECT
  ELSE                              ' If they choose ESC, go back to module level.
    CALL DrawScreen(TheTable)
    CALL ShowRecord(TablesRec)
    Alert$ = "You've escaped. " + KEYSMESSAGE
    CALL ShowMessage(Alert$, 0)
    SLEEP
    Operand$ = ""
  END IF
    CALL EraseMessage
    CALL DrawScreen(TheTable)
    CALL Indexbox(TablesRec, CheckIndex%(TablesRec, FALSE))
```

```
 IF EOF(TablesRec.TableNum) THEN
  Alert$ = "Sorry,  unable to match value you entered with any field value"
  CALL ShowMessage(Alert$, 0)
 END IF

EXIT SUB

SeekHandler:
 IF ERR = 83 THEN                                  ' 83 = Index not found
    CALL DrawScreen(TablesRec.TableNum)
    Alert$ = "SETINDEX for " + IndexName$ + " failed. Need more buffers?"
    CALL ShowMessage(Alert$, 0)
    EXIT SUB
 END IF

END SUB    ' End of SeekRecord procedure

DEFINT A-Z
'******************************** ValuesOK FUNCTION ***************************
'* The ValuesOK FUNCTION checks the values input by the user for various   *
'* purposes. The checking is very minimal and checks the format of what is *
'* entered. For example, the IDnum field needs a double value, but the form*
'* (5 digits, followed by a decimal point, followed by 4 digits) is more   *
'* important than the data type.                                           *
'*                                                                         *
'*                                  Parameters:                            *
'*    Big Rec       User-defined type containing all table information      *
'*    Key1, Key2    Represent strings to check                             *
'*    ValueToSeek   Represents the final value of a combined index         *
'*************************************************************************
FUNCTION ValuesOK (BigRec AS RecStruct, Key1$, Key2$, ValueToSeek$)
  IndexName$ = BigRec.WhichIndex
  ValueToSeek$ = LTRIM$(RTRIM$(ValueToSeek$))
  SELECT CASE RTRIM$(LTRIM$(IndexName$))
    CASE "TitleIndexBS", "PubIndexBS"          ' LEN <= 50
      IF LEN(Key1$) > 50 THEN ValuesOK = FALSE: EXIT FUNCTION
    CASE "AuthorIndexBS", "NameIndexCH"        ' LEN <= 36
      IF LEN(Key1$) > 36 THEN ValuesOK = FALSE: EXIT FUNCTION
    CASE "StateIndexCH"                        ' LEN = 2
      IF LEN(Key1$) > 2 THEN ValuesOK = FALSE: EXIT FUNCTION
    CASE "IDIndex", "IDIndexBO"                ' 5 digits before d.p., 4 after
      IF LEN(ValueToSeek$) <> 10 THEN ValuesOK = FALSE: EXIT FUNCTION
      IF MID$(ValueToSeek$, 6, 1) <> "." THEN
        ValuesOK = FALSE: EXIT FUNCTION
      END IF
```

```
    CASE "CardNumIndexCH", "CardNumIndexBO" ' 5 digits, value <= LONG
      IF LEN(ValueToSeek$) <> 5 THEN ValuesOK = FALSE: EXIT FUNCTION
    CASE "ZipIndexCH"                       ' 5 digits, value <= LONG
      IF LEN(ValueToSeek$) <> 5 THEN ValuesOK = FALSE: EXIT FUNCTION
    CASE "BigIndex"                         ' Key1$ <= 50, Key2$ <= 36
      IF LEN(Key1$) > 50 THEN ValuesOK = FALSE: EXIT FUNCTION
      IF LEN(Key2$) > 36 THEN ValuesOK = FALSE: EXIT FUNCTION
      IF MID$(ValueToSeek$, 6, 1) <> "." THEN
        ValuesOK = FALSE: EXIT FUNCTION
      END IF
  END SELECT
  ValuesOK = TRUE
END FUNCTION

DEFINT A-Z
'******************************** GetOperand FUNCTION ************************
'*                                                                           *
'* The GetOperand FUNCTION displays a choice of operators to allow user a    *
'* choice in how a SEEKoperand search will be conducted. If the user makes   *
'* a valid choice, it is assigned to HoldOperand. An invalid choice or a     *
'* choice of ESC results in "<>" being passed back. This permits an exit     *
'* from the function (which is recursive). Otherwise, the user's choice is   *
'* trapped in HoldOperand when ENTER is pressed.                             *
'* Note: since  this function is recursive so use the calls menu to keep     *
'* track of the nesting depth when stepping through it. Unlike PlaceCursor   *
'* GetOperand doesn't keep track of the stack - the stack set should be OK.  *
'*                                                                           *
'*                              Parameters:                                  *
'*   HoldOperand    Contains operand to check each time function calls       *
'*                  itself; Let's user ESC from function if desired.         *
'*                                                                           *
'*****************************************************************************
FUNCTION GetOperand% (HoldOperand$)
  STATIC WhichOne     ' Keep track of which case from call to call

  ' If user has chose ESC then exit back to caller
  IF HoldOperand$ = "<>" THEN WhichOne = 0: EXIT FUNCTION

  ' If this is the first time through the function then
  ' replace the Sort Order box with box of operand choices.
  IF WhichOne = 0 THEN
    RESTORE OperandBox
    FOR Row = BOXTOP TO BOXEND
      LOCATE Row, 42
      READ Temp$
      PRINT Temp$
```

```
      IF Row = BOXEND THEN
        COLOR FOREGROUND + BRIGHT, BACKGROUND
        LOCATE Row, INDBOX + 5
        PRINT "Relationship to Key"
      END IF
    NEXT Row
    LOCATE VLINE, 44
    PRINT "Equal To     Value Entered"      ' This is default --- if user
    COLOR FOREGROUND, BACKGROUND            ' presses Enter without tabbing,
  END IF                                    ' SeekRecord sets the operand
                                            ' to =. Note: a more flexible
                                            ' default choice might be >=.

Alert$ = "Now press TAB to select how search should be conducted"
CALL ShowMessage(Alert$, 0)
DO
Answer$ = INKEY$
LOOP WHILE Answer$ <> CHR$(TABKEY) AND Answer$ <> CHR$(ENTER)_
AND Answer$ <> CHR$(ESCAPE)

IF LEN(Answer$) = 1 THEN
  SELECT CASE ASC(Answer$)
    CASE TABKEY
      SELECT CASE WhichOne
        CASE 0
          COLOR FOREGROUND, BACKGROUND
          LOCATE VLINE, 44
          PRINT "Equal To"
          COLOR BRIGHT + FOREGROUND, BACKGROUND
          LOCATE RLINE, 44
          PRINT "Greater Than"
          WhichOne = WhichOne + 1
          HoldOperand$ = ">"
CASE 1
   COLOR BRIGHT + FOREGROUND, BACKGROUND
   LOCATE VLINE, 44
   PRINT "Equal To"
   'COLOR BRIGHT + FOREGROUND, BACKGROUND
   LOCATE WLINE, 44
   PRINT "or"
   WhichOne = WhichOne + 1
   HoldOperand$ = ">="
```

```
    CASE 2
      COLOR FOREGROUND, BACKGROUND
      LOCATE RLINE, 44
      PRINT "Greater Than"
      LOCATE WLINE, 44
      PRINT "or"
      COLOR BRIGHT + FOREGROUND, BACKGROUND
      LOCATE ALINE, 44
      PRINT "or"
      LOCATE ELINE, 44
      PRINT "Less Than"
      WhichOne = WhichOne + 1
      HoldOperand$ = "<="
    CASE 3
      COLOR FOREGROUND, BACKGROUND
      LOCATE VLINE, 44
      PRINT "Equal To"
      LOCATE ALINE, 44
      PRINT "or"
      WhichOne = WhichOne + 1
      HoldOperand$ = "<"
      SLEEP
    CASE 4
      COLOR FOREGROUND, BACKGROUND
      LOCATE ELINE, 44
      PRINT "Less Than"
      COLOR BRIGHT + FOREGROUND, BACKGROUND
      LOCATE VLINE, 44
      PRINT "Equal To     Value Entered"
      WhichOne = WhichOne + 1
      HoldOperand$ = "="
    CASE ELSE
  END SELECT                              ' If no choice was made, call
    IF WhichOne > 4 THEN WhichOne = 0     ' GetOperand again.
    COLOR FOREGROUND, BACKGROUND
    OK = GetOperand%(HoldOperand$)
  CASE ENTER
    WhichOne = 0
    EXIT FUNCTION
CASE ESCAPE                             ' If user chooses ESC, signal the function
  HoldOperand$ = "<>"                   ' to exit and keep exiting back through
  GetOperand% = 0                       ' all levels of recursion
  WhichOne = 0
CASE ELSE                               ' If user chooses invalid key, try again.
  BEEP
  CALL ShowMessage("Use TAB to select relationship to search for...", 0)
```

```
        COLOR WHITE, BACKGROUND
        OK = GetOperand%(HoldOperand$)
    END SELECT
  ELSE
END IF

END FUNCTION
```

A Multi-Table Database

So far, only the BookStock table has been shown in the BOOKS.MDB database. Even in the database of a hypothetical library, it would make sense to have another table in BOOKS.MDB to hold information about each library-card holder and a third table to hold information that relates specific copies of books to card holders who may have borrowed them. In fact, CardHolders is a reasonable name for one table, and BooksOut will do for the other. Figure 10.9 illustrates a possible design for the CardHolders table. Figure 10.10 illustrates the BooksOut table.

Card Holders

CardNum	The Name	Street	City	State	Zip
31277	Button, Tom	WhereEver Road	Kirkland	WA	98042
17445	Burke, John	232 NE Happy Way	Redmond	WA	98053
.					
.					
.					
23175	Grimm, Rick	Somewhere Place	Bellevue	WA	98021

Figure 10.9 The CardHolders Table

The user-defined type by which a program gains access to the CardHolders table looks as follows:

```
TYPE            CardHolders
   CardNum      AS INTEGER
   Zip          AS LONG
   TheName      AS STRING * 36
   City         AS STRING  *  26
   Street       AS STRING  *   50
   State        AS STRING * 2
END TYPE
```

The user-defined type by which a program gains access to the BooksOut table can have element names that duplicate those in other tables, since they are accessed using dot notation. The `OutBooks` type looks as follows:

```
TYPE BooksOut
   IDnum      AS DOUBLE
   CardNum    AS LONG
   DueDate    AS DOUBLE
END TYPE
```

BooksOut

IDnum	CardNum	DueDate
81200.4253	31277	32839
67202.2683	17445	32863
.		
.		
.		
13351.3501	23175	32921

Figure 10.10 The BooksOut Table in BOOKS.MDB

When these tables are added, BOOKS.MDB contains three tables. The BookStock table is related to the CardHolders table through the IDnum column in the BooksOut table. For inventory purposes, you might manipulate just the BookStock table, and for doing a mailing of new-titles notices you could manipulate only the CardHolders table. To find out which card holder has withdrawn which copy of a particular book, you can get the `IDnum` from the BookStock table, then look up that `IDnum` in the BooksOut table. If the book was overdue, you could get the `CardNum` value from the BooksOut table, then look up that card number in the CardNum column in the CardHolders table to get information about the borrower.

Book Stock

Edition	IDnum	Price	Title	Publisher	Author
1	15561.1276	22.95	QuickBASIC ToolBox	Microsoft Press	Craig, J.C.
2	15561.5125	19.95	QuickBASIC V. 4.0	Microsoft Press	Herger, D.
.	.				
.	.				
.	.				
1	47161.3011	35.00	Using MS QuickBASIC	Wiley	Cooper, J.

BooksOut

IDnum	CardNum	DueDate
81200.4253	31277	32839
67202.2683	17445	32863
.	.	
.	.	
.	.	
13351.3501	23175	32921

Card Holders

CardNum	Zip	The Name	City	Street	State
17445	98053	Burke, John	Redmond	232 NE Happy Way	WA
31277	98042	Button, Tom	Kirkland	WhereEver Road	WA
31277	98021	Grimm, Rick	Bellevue	Somewhere Place	WA

Figure 10.11 Relationship of Tables in BOOKS.MDB Database

Example

As your librarian traverses the BookStock table, he or she might want to check the due date on some of the books. Pressing the W key calls the `GetStatus` procedure to look up the book ID number in the BooksOut table and retrieve the corresponding `BooksOut` record. The `ShowStatus` procedure is called to display the due date of the book. Note that `ShowStatus` currently displays the date in raw serial form, as it appears in the table. To convert a serial date for normal display, you can replace the expression `STR$(ValueToShow)` with appropriate calls to the date and time function library (DTFMTER.QLB), supplied with Microsoft BASIC.

BOOKLOOK.BAS contains other routines that use information from all the tables to automate library procedures. For example, the `BooksBorrowed` procedure is accessible when the CardHolders table is being displayed. If the user presses B (for Books Outstanding), `BooksBorrowed` compiles a list of the books checked out to that card holder. `LendeeProfile` gives information on the borrower of the title currently displayed from the BookStock table. The `BorrowBook` procedure (not shown in the following listing) allows a book to be checked out simply by typing in the user's name. If the user is not a valid card holder, a warning is displayed. If the user has a library card, `BorrowBook` displays the card

holder's information so it can be checked to see what the due date will be, and also to see if the personal information needs to be updated. The `ReturnBook` procedure (not shown in the following listing) displays the name of the borrower, and calculates and displays any fines that may be due.

```
'       Module-level code using relationships among tables.
.
.
.
' STATUS gets the due date of a book & displays it.
CASE STATUS
  IF BigRec.TableNum = cBookStockTableNum THEN
    CALL ShowStatus("", 0#)                      ' Explicitly type the 0
    GotIt = GetStatus(BigRec, DateToShow#)       ' to avoid type mismatch
    IF GotIt THEN
      Alert$ = "Press B for information on Borrower of this book"
      CALL ShowMessage(Alert$, 0)
      CALL ShowStatus("Due Date: ", DateToShow#)
      SLEEP: EraseMessage
    END IF
  END IF

 ' LendeeProfile displays borrower of displayed book.
CASE BORROWER
  CALL LendeeProfile(BigRec)

' BooksBorrowed shows books borrowed by CardHolder.
CASE WHICHBOOKS
  CALL BooksBorrowed(BigRec)

' If user hits "V" cycle through displayable tables.
CASE OTHERTABLE
  IF BigRec.TableNum < cDisplayedTables THEN
    BigRec.TableNum = BigRec.TableNum + 1
  ELSE
    BigRec.TableNum = 1
  END IF
  EXIT DO

' If user picks "I" to check current book back in, make sure it is out,
' then check it back in.
CASE CHECKIN
    GotIt = GetStatus(BigRec, DateToShow#)
  IF DateToShow# THEN                            ' ReturnBook is
    CALL ReturnBook(BigRec, DateToShow#)         ' not shown in
  END IF                                         ' this listing
```

```
    ' If user picks "O" to check current book out, make sure it is available,
    ' then check it out
    CASE CHECKOUT
        GotIt = GetStatus(BigRec, DateToShow#)          ' BorrowBook is
      IF DateToShow# = 0# THEN                          ' not shown in
        CALL BorrowBook(BigRec)                         ' this listing
      ELSE
        CALL ShowMessage("Sorry, this book is already checked out...", 0)
        SLEEP: EraseMessage
      END IF
     .
     .
     .
DEFINT A-Z
'******************************** GetStatus FUNCTION ************************
'*  The GetStatus FUNCTION looks up the status of a book in the BooksOut    *
'*  table. If the SEEK fails it means the book isn't checked out, and that  *
'*  message is displayed. Otherwise, it is placed in DateToShow parameter.  *
'*  The final message about retrieving borrow info relates to LendeeProfile.*
'*                                                                          *
'*                              Parameters:                                 *
'*  TablesRec     Structure containing the information about all the tables *
'*  DateToShow    The due date to show in the ShowStatus SUB                *
'*****************************************************************************
FUNCTION GetStatus (TablesRec AS RecStruct, DateToShow#)
  IF GETINDEX$(cBooksOutTableNum) <> "IDIndexBO" THEN
    SETINDEX cBooksOutTableNum, "IDIndexBO"
  END IF
  SEEKEQ cBooksOutTableNum, TablesRec.Inventory.IDnum
  IF NOT EOF(cBooksOutTableNum) THEN
    RETRIEVE cBooksOutTableNum, TablesRec.OutBooks
  ELSE
    Alert$ = "This book is not checked out"     ' the book wasn't in BooksOut
    CALL ShowMessage(Alert$, 0)                 ' table, so it wasn't out
    SLEEP: CALL EraseMessage
    DateToShow# = 0: GetStatus = FALSE
    EXIT FUNCTION
  END IF
  DateToShow# = TablesRec.OutBooks.DueDate#
  GetStatus = TRUE
END FUNCTION
```

```
DEFINT A-Z
'******************************** ShowStatus SUB ****************************
'* The ShowStatus SUB uses the due date associated with the book IDnum from*
'* of the BooksOut table. This date is in serial form which is not decoded *
'* here, but can be decoded with the date/time function library supplied   *
'* with BASIC. The due date is displayed centered on the top line of       *
'* the ShowMessage box.                                                    *
'*                                  Parameters:                            *
'*  Stat$       Message introducing the due date when displayed in its box *
'*  ValueToShow The due date of the book from the BooksOut table           *
'************************************************************************
SUB ShowStatus (Stat$, ValueToShow AS DOUBLE)
COLOR FOREGROUND, BACKGROUND
DataEndLine$ = STRING$(60, 205)        'Redraw the bottom line.
StringToShow$ = Stat$          ' Figure out where to locate the text.
IF ValueToShow = 0 THEN
  LOCATE TABLEEND, 4
  PRINT DataEndLine$
  EXIT SUB
ELSE
  ' The dates in the file are in serial form. Use the date/time library
  ' to decode serial dates for normal display. In the code below, you
  ' could just substitute STR$(ValueToShow) with a call to the library.

  StringToShow$ = StringToShow$ + STR$(ValueToShow)
  HowLong = LEN(StringToShow$)
  PlaceStatus = (73 \ 2) - (HowLong \ 2)
  StatusSpace$ = CHR$(181) + STRING$(HowLong, 32) + CHR$(198)
END IF
LOCATE TABLEEND, PlaceStatus
PRINT StatusSpace$
COLOR BACKGROUND, BRIGHT + FOREGROUND
LOCATE TABLEEND, PlaceStatus + 1
PRINT StringToShow$
COLOR FOREGROUND, BACKGROUND
END SUB

DEFINT A-Z
'****************************** LendeeProfile SUB ***************************
'*  The LendeeProfile takes the IDnum of the currently displayed book, then*
'*  looks that up in the BooksOut table and fetches the CardHolder record  *
'*  that corresponds to the CardNum entry in BooksOut. Then the CardNum is *
'*  looked up in the CardHolders table and the borrower information shown. *
'*                                  Parameters                             *
'*  TablesRec   Contains information on all the tables in the database     *
'************************************************************************
```

```
SUB LendeeProfile (TablesRec AS RecStruct)
  ' Create an array to hold information from CardHolders table.
  DIM LendeeInfo(10)  AS STRING
  ' Set the index if it is not the one you want.
  IF GETINDEX$(cBooksOutTableNum) <> "IDIndexBO" THEN
    SETINDEX cBooksOutTableNum, "IDIndexBO"
  END IF
  SEEKEQ cBooksOutTableNum, TablesRec.Inventory.IDnum     ' Seek the record.
  IF EOF(cBooksOutTableNum) THEN                          ' If you find it,
    CALL ShowMessage("This book is not checked out", 0)   ' the book is out,
    EXIT SUB                                              ' otherwise not.
  ELSE                                                    ' If it's there,
    RETRIEVE cBooksOutTableNum, TablesRec.OutBooks        ' fetch it.

' If the CardNum exists, set an index in CardHolders and SEEK the
' CardNum. If SEEK fails, print a warning; if it succeeds, get the
' information about the borrower, and display it using PeekWindow

IF TablesRec.OutBooks.CardNum <> 0 THEN
  IF GETINDEX$(cCardHoldersTableNum) <> "CardNumIndexCH" THEN
    SETINDEX cCardHoldersTableNum, "CardNumIndexCH"
  END IF
  SEEKEQ cCardHoldersTableNum, TablesRec.OutBooks.CardNum
  IF EOF(cBooksOutTableNum) THEN
    Alert$ = "Cardholder number associated with book ID is not valid"
    CALL ShowMessage(Alert$, 0)
    EXIT SUB
  ELSE
    RETRIEVE cCardHoldersTableNum, TablesRec.Lendee
    LendeeInfo(0) = RTRIM$(TablesRec.Lendee.TheName)
    LendeeInfo(1) = ""
    LendeeInfo(2) = RTRIM$(TablesRec.Lendee.Street)
    LendeeInfo(3) = RTRIM$(TablesRec.Lendee.City)
    LendeeInfo(4) = RTRIM$(TablesRec.Lendee.State)
    LendeeInfo(5) = LTRIM$(STR$(TablesRec.Lendee.Zip))
    LendeeInfo(7) = STR$(TablesRec.Lendee.CardNum)
    LendeeInfo(6) = ""
    LendeeInfo(7) = "Card number: " + LendeeInfo(7)
    LendeeInfo(8) = ""
    FOR Index = 1 TO 6
      ThisBig = LEN(LendeeInfo(Index))
      IF ThisBig > BiggestYet THEN
        BiggestYet = ThisBig
      END IF
    NEXT Index
    Alert$ = "Press V to access the record for this cardholder"
```

```
        CALL ShowMessage(Alert$, 0)
        HeadMessage$ = "Borrower of this Book"
        FootMessage$ = "Press a key to clear box"
        CALL ClearEm(TablesRec.TableNum, 1, 1, 1, 1, 1, 1)
        CALL PeekWindow(LendeeInfo(), HeadMessage$, FootMessage$, BiggestYet, 9)
        CALL DrawTable(TablesRec.TableNum)
        CALL ShowMessage(KEYSMESSAGE, 0)
      END IF
    END IF
  END IF
END SUB

DEFINT A-Z
'********************************** BooksBorrowed SUB ************************
'*  The BooksBorrowed SUB takes the CardNum in BooksOut associated with the*
'*  currently displayed CardHolder, then looks up each book in BooksOut    *
'*  assigned to that CardNum. Note that you can use SEEKoperand to find the*
'*  first matching record, but thereafter you need to MOVENEXT and check   *
'*  each succeeding record to see if the CardNum matches. When a match is  *
'*  made, look up the IDnum in the BooksOut table and retrieve the title.   *
'*  Put all the titles in the Titles array, then display with PeekWindow.   *
'*                                                                         *
'*                                  Parameters:                            *
'*  TablesRec   Structure containing information on all database tables    *
'***************************************************************************
SUB BooksBorrowed (TablesRec AS RecStruct)
  DIM Titles(50) AS STRING
  ' First, get the card number of the current record in BookStock; then,
  ' at the end of this procedure, restore that book
  IF GETINDEX$(cBooksOutTableNum) <> "CardNumIndexBO" THEN
    SETINDEX cBooksOutTableNum, "CardNumIndexBO"
  END IF
  RevName$ = TransposeName$(TablesRec.Lendee.TheName)
SEEKEQ cBooksOutTableNum, TablesRec.Lendee.CardNum
 IF NOT EOF(cBooksOutTableNum) THEN
  DO
    RETRIEVE cBooksOutTableNum, TablesRec.OutBooks
     IF TablesRec.OutBooks.CardNum = TablesRec.Lendee.CardNum THEN
        IF GETINDEX$(cBookStockTableNum) <> "IDIndex" THEN
          SETINDEX cBookStockTableNum, "IDIndex"
        END IF
        SEEKEQ cBookStockTableNum, TablesRec.OutBooks.IDnum
        IF NOT EOF(cBookStockTableNum) THEN
          RETRIEVE cBookStockTableNum, TablesRec.Inventory
          Titles(Index) = RTRIM$(TablesRec.Inventory.Title)
          ThisSize = LEN(RTRIM$(Titles(Index)))
```

```
            IF ThisSize > Biggest THEN
              Biggest = ThisSize
            END IF
          Index = Index + 1
        END IF
      END IF
    MOVENEXT cBooksOutTableNum
    LOOP UNTIL EOF(cBooksOutTableNum)
  ELSE
    Alert$ = RevName$ + " currently has no books checked out"
    CALL ShowMessage(Alert$, 0)
  END IF
  IF Index <> 0 THEN
    HeadMessage$ = " Books borrowed by " + RevName$ + " "
    FootMessage$ = " Press a key to continue "
    CALL PeekWindow(Titles(), HeadMessage$, FootMessage$, Biggest, Index)
    CALL DrawTable(TablesRec.TableNum)
    CALL ShowMessage(KEYSMESSAGE, 0)
  END IF
END SUB
```

Deleting Indexes and Tables

The **DELETEINDEX** and **DELETETABLE** statements let you delete an index or a table from the database. These statements have the following syntax:

DELETEINDEX *filenumber%, indexname$*

DELETETABLE *database$, tablename$*

Argument	Description
filenumber%	A numeric expression representing the table on which the index to be deleted was created.
indexname$	A string expression representing the name of the index to be deleted.
database$	A string expression representing the name of the database containing the table to be deleted.
tablename$	A string expression representing the name of the table to be deleted.

It isn't always easy to anticipate which columns a user may want to index when using a database. As an advanced feature of a program you might want to let a user create indexes during run time. If so, **DELETEINDEX** can be used to remove the specified index from the database when it is no longer needed.

You can use the **DELETETABLE** statement to delete an old table when you no longer need any of its data records. Once you delete a table and close the database, there is no way to

recover the records. You should assume that the table is deleted from the file immediately upon execution of this statement, rather than at some future time (for example, when the file is closed). Note that when you delete a table, all information (including indexes, etc.) in the data dictionary relating to the table is also deleted.

You cannot practically write routines to permit a user to create custom tables during run time, since the user-defined type that describes a table must already exist when the program begins. Although it isn't covered here, you can write routines that permit a user to create a table that duplicates the columns of a table already in the database. For example, you might want to permit copying of a subset of a table's records to a table of the same type.

Executing **DELETETABLE** or **DELETEINDEX** commits any pending transactions.

ISAM Naming Convention

Some parts of an ISAM database require names (for example, tables, columns, and indexes), and these names must conform to the ISAM naming convention. The ISAM convention is essentially a subset of the BASIC convention, as shown in the following table:

ISAM convention	BASIC convention
30 characters or fewer.	40 characters or fewer.
Alphanumeric characters only, including A–Z, a–z, and 0–9.	Alphanumeric characters, plus the BASIC type-declaration characters, where appropriate (variables and functions).
Must begin with alphabetic character.	Must begin with alphabetic character, but only **DEF FN** functions can begin with "fn."
No special characters allowed.	The period is not allowed in the names of elements within a user-defined type. Since these are the names BASIC and ISAM have in common, there is no conflict.
Not case sensitive.	Not case sensitive.

Starting ISAM for Use in QBX

Microsoft BASIC includes two terminate-and-stay-resident (TSR) programs, PROISAM.EXE and PROISAMD.EXE. You can use either of these programs to place the ISAM routines in memory when you develop or run database programs within the QBX environment. The benefit of this approach is that, when you are working on programs or modules that don't need ISAM, the amount of memory required by QBX is substantially reduced.

PROISAM.EXE contains all ISAM routines needed to run most database programs. It does not contain the "data dictionary" statements —**CREATEINDEX**, **DELETEINDEX**, and **DELETETABLE**. It contains a restricted version of the **OPEN...FOR ISAM** statement that will open a database or table, but will not create it if it does not already exist. Since you often will not need to program the creation and deletion of indexes and tables within an end-user program, PROISAM.EXE is usually sufficient.

PROISAMD.EXE contains all the ISAM routines.

You use the following syntax to start either PROISAM.EXE or PROISAMD.EXE:

{PROISAM | PROISAMD} [[/Ib:*pagebuffers*]] [[/Ie:*emsreserve*]] [[/Ii:*indexes*]] [[/D]]

Argument	Description
/Ib:*pagebuffers*	Increases the amount of conventional memory reserved for ISAM's buffers. The defaults are 6 *pagebuffers* (12K of 2K pages) for PROISAM, and 9 *pagebuffers* (18K of 2K pages) for PROISAMD. There is also 3–5K used for data by PROISAM and 14–16K for PROISAMD. Maximum value for *pagebuffers* is 512. However, since DOS only provides 640K, this maximum is not possible in conventional memory. Determine the optimal value for a specific program by experimentation. Note that the default values for *pagebuffers* are the minimums necessary for an ISAM program, not the optimal or average values. If you (or your anticipated users) don't have EMS, more than the default number of *pagebuffers* may be necessary for running the program. Even if a program runs with the default number of *pagebuffers*, specifying the most buffers possible improves ISAM performance. Note, however, that if no EMS is available, too high a value for /Ib could allocate so much memory to ISAM buffers that the TSR would not be able to remove itself from memory when invoked with the /D option. When EMS is available, the amount specified with /Ib is taken first from EMS, and if that is not sufficient, the rest is taken from conventional memory.
/Ie: *emsreserve*	If you have expanded memory, ISAM will automatically use up to 1.2 megabytes of it for buffers (that is, it will place up to 512 2K *pagebuffers*, plus about 10 percent overhead space, into EMS). ISAM takes as much EMS memory as possible by default, which frees conventional memory for other uses while improving performance. The number given in the /Ie option lets you specify how much expanded memory should be reserved for purposes other than ISAM. This limits the amount of EMS that ISAM uses for *pagebuffers*, since ISAM will take only EMS between the *emsreserve* specified and the total EMS available. The default value for *emsreserve* is 0. State values in kilobytes; for example, `/Ie:500` specifies 500K should be left available for other purposes. In practice, you only need to specify /Ie if your program code (or a loaded Quick library) actually manages EMS memory. In such a case, you should also specify the /Es option when starting QBX or when compiling with BC. Specifying a value of –1 for /Ie reserves all EMS for other uses, and none is used by ISAM.

/Ii: *indexes*	Specifies the number of non-NULL indexes used in the program. Use this option if your program has more than 30 indexes. If you don't specify a value for this option, ISAM assumes your database contains no more than 30 defined indexes. If this value is too low, your program may fail. Maximum permissible value is 500.
/D	Removes the ISAM TSR from memory.

Note

When you use transactions, ISAM keeps a transaction log. The name of the log file is guaranteed to be unique, so multiple ISAM programs can be run in simultaneous windows in an operating environment like Windows 386 without the danger of conflicts within the log files. In versions of DOS earlier than 3.0 however, the name of the file is ~PROISAM.LOG, and it is created in the /TMP directory by default; otherwise it goes in the current directory. If you set your /TMP environment variable to a RAM drive, transaction logging will be faster. Don't confuse this log file with the PROISAM.EXE TSR. If a log file appears in your current directory, you can delete it; ISAM overwrites the old log file each time a transaction is initiated.

Estimating Minimum ISAM Buffer Values

When you load the ISAM TSR, it requires memory beyond its disk-file size because it reserves a certain amount of memory for buffers in which it does most of its work. The actual amount of buffer space reserved depends on which version of the TSR you are using, and the number you specify for the /Ib and /Ii options. The defaults represent the absolute minimum required for a minimal ISAM program. Having more buffers available always improves performance, and in some cases may be necessary for a program to run at all. Additional buffers improve performance because, when the buffers are full, some of their contents is written back to the disk. This causes a disk access to update the disk file, and another access when the material that was swapped out has to be swapped back in. The swapping system is based on a least-recently-used (LRU) algorithm. The more buffers that are available, the less likely it is that any particular piece of material will need to be swapped in or out. To get a basic idea of the minimum number of buffers your program needs, use the maximum of 9 or 6 (the default buffer settings, depending on whether you use PROISAM or PROISAMD), or the following formula:

$pagebuffers = 1 + w + x + 4y + 8z$

In the preceding formula:

w = the maximum number of open tables containing data

x = total of non-NULL indexes used in the program

y = 1, if **INSERT** or **UPDATE** statements are executed, otherwise 0

z = 1, if a **CREATEINDEX** statement is executed, otherwise 0

Depending on the density of ISAM statements in any section of code, it is possible that the default number of buffers will not be adequate to handle the necessary processing.

Any EMS (up to 1.2 megabytes) that is available is used for ISAM buffers. This leaves an equivalent amount of conventional memory for other purposes. Note however, that only the ISAM buffers are placed in EMS, the ISAM code (represented approximately by the disk-file size) itself resides in conventional memory. Use the /Ie option to reserve any EMS that may be needed when your program actually manages EMS internally, or works concurrently with any other programs that use EMS.

ISAM and Expanded Memory (EMS)

As noted above, ISAM uses conventional and expanded memory as long as the expanded memory conforms to the Lotus-Intel-Microsoft Expanded Memory Specification (LIM 4.0). Using expanded memory correctly can enhance both the performance and capacity of programs. The actual management of expanded memory is done for you by BASIC within the limits you set with the /Ie ISAM option and the /Es QBX option. If expanded memory is available, ISAM uses the difference between the total and the amount you specify with the /Ie option, up to a maximum of about 1.2 megabytes.

There are several factors to consider in using the /Ib and /Ie ISAM options (and the /Es and /Ea QBX options), including the following:

- The system on which you develop a program may have different memory resources than the system on which your user runs the program.
- If your program performs explicit EMS management, or uses a library that does, you need to reserve the necessary EMS when starting the TSR or compiling the program. QBX and BC have options dealing with EMS (/Es and /Ea) that interact with ISAM's use of EMS in certain situations. You may need to use the /Es option when invoking QBX or BC.

In order to provide for users who may not have EMS, you should always specify an optimal setting for the /Ib ISAM option. Beyond this, in most cases, it should suffice to allow the defaults to determine the apportioning of EMS between ISAM and other EMS usage. The EMS defaults are designed to make the best use of whatever combinations of conventional memory and EMS may be available. In general, trying to optimize values between /Ib and /Ie only makes sense if your program itself actually performs expanded memory management, in which case you should be sure there is enough EMS available for it at run time.

For example, if you want ISAM to use 22 buffers, the buffers require 44K (each buffer requires 2K) plus up to 5K for data in PROISAM.EXE, and up to 16k for data with PROISAMD.EXE. The 22 buffers plus overhead for PROISAM.EXE will require about 49K. However, if EMS is available and you don't specify a value for /Ie, ISAM will use one megabyte of expanded memory. If you have only one megabyte of expanded memory available, and you have written your program to explicitly manage 500K of that, you need to specify 500 as a value for the /Ie option. This reserves the amount of expanded memory your program manages. If an end user of the program has no EMS available beyond the 500K you have reserved, all the memory needed for the ISAM *pagebuffers* is taken from conventional memory.

The actual algorithm used by ISAM for apportioning buffers and EMS is as follows:

1. Reserve EMS as specified by /Ie. If less EMS is available than is specified, reserve all EMS.

2. Allocate non-buffer memory needed by ISAM. Take this first from available EMS, then, if that is insufficient, take the remainder from conventional memory.
3. Allocate the number of buffers specified by the /Ib option, first from EMS, then, if that is insufficient, from conventional memory.
4. If more EMS is available than was needed to satisfy /Ib, keep allocating buffers from EMS until all EMS is consumed, or until the ISAM limit is reached (512 buffers, or about 1.2 megabytes including overhead).
5. Release the EMS reserved in step 1 (by the /Ib option) for use by other programs.

When you create an executable file from within QBX, whether or not the program will need to have the TSR invoked depends on options you chose for ISAM during Setup, as explained in the next section.

Using ISAM with Compiled Programs

There are several types of executable files you can produce for ISAM programs, depending on your needs and the needs of your users. Programs that require the presence of a TSR make sense if you are distributing several distinct ISAM-dependent programs on the same disk. Each individual program could be significantly smaller if all made use of the ISAM routines from the TSR. If you compile programs so they need the run-time module, you can have the ISAM routines linked in as part of the run-time module, or have the user start one of the TSR programs before starting the application.

In any of these cases, you have a choice of TSRs. PROISAMD.EXE contains all the ISAM routines, including the data dictionary routines (for creating and deleting indexes, tables, databases, etc.). At various times, not all the routines are necessary. For example, if you create a program and supply an empty database file (one with tables and indexes, but no data records), your program would have no real need to create tables or indexes, since they would already exist in the file. In such cases, you could supply the smaller TSR program (called PROISAM.EXE) to conserve memory. Table 10.2 describes the requirements for various ISAM configurations.

Table 10.2 Executable File for ISAM Programs

Program type	Must supply	Compile with	Libraries to link
Case 1: Stand-alone executable	No ISAM TSR	BC using /O option	BCL70*mso*.LIB
Case 2: Stand-alone executable requiring the ISAM TSR	PROISAM.EXE or PROISAMD.EXE	BC using /O option	BCL70*mso*.LIB
Case 3: Executable requiring the BASIC Run-time Module	No ISAM TSR	BC without the /O option	BRT70*mso*.LIB

Table 10.2 *Continued*

Program type	Must supply	Compile with	Libraries to link
Case 4: Executable requiring the BASIC Run-time Module and the ISAM TSR	PROISAM.EXE or PROISAMD.EXE	BC without the /O option	BRT70*mso*.LIB

When you ran the Setup program, you had the opportunity to choose libraries that would create executable files containing all the ISAM routines. You could also choose an option that created run-time modules that contained all the ISAM routines. If you didn't choose these options, your executable files (and run-time modules) will require your users to run either PROISAM.EXE or PROISAMD.EXE before using the database programs. To qualify for Case 1 in Table 10.2, you should have specified the full ISAM or reduced ISAM option for BCL70*mso*.LIB. Then, when the executable file is created, you need to have either PROISAMD.LIB or PROISAM.LIB in your current directory or library search path. For Case 2 you should have chosen "ISAM Routines in TSR" during Setup. In this case you don't need to have PROISAM.LIB or PROISAMD.LIB accessible when the executable file is created. You made the same selections regarding run-time modules during Setup (cases 3 and 4).

When you compile a stand-alone program from the command line (that is, one that does not require the presence of the TSR at run time), you can use /Ib, /Ie, and /Ii as options to the BC command. Their syntax and general effects are the same with the stand-alone program as described in the previous section. If your program uses run-time overlays, the EMS is automatically allocated for the overlays first, before ISAM. If you don't want overlays to use EMS you can link the program with NOEMS.OBJ and overlays will be swapped to disk instead. If overlays use EMS, ISAM will take whatever remains after EMS allocation for the overlays—up to 1.2 megabyte. If your program does internal EMS memory management, it can only be done from within a non-overlayed module. However, in such a case, you should probably link with NOEMS.OBJ. Also, remember to compile with the /Es option to BC. You should always specify a correct value for /Ii if your program uses more than 30 indexes.

However, note that sharing expanded memory between ISAM and other uses inhibits ISAM performance, since the ISAM buffers and other EMS usage must use the same EMS window to access the expanded memory. This means that with each call to the ISAM library, the EMS state must be saved and restored. If you must share EMS memory between ISAM and other things, use the relative amounts that optimize ISAM performance. In such cases, use the /Es option to guarantee EMS save and restore with mixed-language code that manages EMS.

Important When you compile a program from within QBX, only the QBX options are passed to the compiler. This is fine if an ISAM stand-alone program will use the TSR, since the buffer and index options are specified when the TSR is invoked. However, if the program is to have the ISAM routines included in the executable file, you must compile the program's main module from the command line, and specify the appropriate /I and /E options as arguments to BC.

Practical Considerations when Using EMS

Note that the ISAM /Ie option and the QBX options /E, /Ea, and /Es have the effect of reserving EMS for programs that use internal EMS management (or other applications), rather than specifically limiting the amount of EMS used by the program for which the option is supplied. ISAM uses EMS to improve performance by radically reducing the frequency of disk access. In general, the automatic apportioning of conventional and EMS memory should cover the widest range of situations best, because with each allocation of EMS, whatever is available is used whenever it can be.

During development of a very large program, it may be more beneficial to reserve most available EMS for QBX (except the minimum ISAM needs for buffers and indexes), since the speed of ISAM is probably not as important as the ability to have QBX place units of code in EMS, thus increasing the potential size of the source files you can fit in QBX. However, since QBX only places units of code in the 0.5–16K range in expanded memory, this is only optimal if your coding style is to use small to moderate code units (**SUB** and **FUNCTION** procedures, and module-level code). In a compiled program, the ISAM performance in the executable file is the most important feature, so compiling with high values for /Ib (just to provide for users with no EMS) and no specification for /Ie should offer the best results. ISAM never uses more than 1.2 megabytes, so all remaining EMS is automatically available for other uses. Such other uses include your program's code units, arrays up to 16K (if /Ea is specified), or explicit EMS management within the program.

Note however, that dividing EMS between ISAM and other uses slows ISAM and QBX performance to some degree. It may make sense during program development, but might not be satisfactory in a compiled program. If you want this type of sharing, use the /Ie option to reserve EMS for the overlays, plus the /Es option to ensure EMS saving and restoration. EMS in a compiled program is automatically used for run-time overlays (if you use them). To prevent EMS sharing, compile with BC without using the /Ie option or /Es, and specify `/E:0` to prevent use of EMS for anything but ISAM. For more information on using the /Es option for QBX, see Chapter 3, "Memory Management for QBX," in *Getting Started*. Run-time overlays are discussed in Chapters 15, "Optimizing Program Size and Speed," and 18, "Using LINK and LIB," of this book.

Note BASIC releases EMS when the program terminates due to a run-time error, as well as an **END**, **STOP**, or **SYSTEM** statement. If a program terminates for some other reason while EMS is being used, that portion of EMS will not be available again until the EMS manager is restarted. If the EMS manager is the one used in Microsoft Windows 386, you can simply exit from Windows, then start Windows again to recover the EMS. If your EMS manager is started by an entry in a CONFIG.SYS file, you may need to reboot to recover use of EMS.

TSRs and Installation/Deinstallation Order

If you (or your users) will be using other TSR programs besides the ISAM TSR, they should be installed before the ISAM TSR. The reason for this is that the ISAM TSR is only needed when the ISAM program is run. If you finish with your ISAM program and have installed another

TSR after ISAM, you will have to remove any more-recently installed TSR programs before you can successfully remove the ISAM TSR. Otherwise, the /D option to the ISAM TSR will remove ISAM from memory, but the memory cannot be used by the other programs, and the operating system may be destabilized. If you attempt to remove TSRs in an improper order, a warning message is displayed.

Block Processing Using Transactions

To accommodate data entry errors, ISAM includes three transaction statements and one transaction function that allow you to restore a database to a previous state. By using these in conjunction with the **CHECKPOINT** statement (which lets you explicitly write all open databases to disk) you can enhance the integrity of your user's databases.

When you use **UPDATE** to change a record in a table, the change is made immediately. However, the actual writing of data to disk is done at periodic intervals determined by the ISAM engine. The **CHECKPOINT** statement requires no argument. It simply forces the current state of all open databases to be written to disk.

Conversely, you can code your program to allow a user (or a routine in the program) to retract a sequence of operations either selectively or as a block. Using transactions (block processing) can help ensure consistency to operations performed on multiple tables and multiple databases.

The following table briefly describes these block-processing statements:

Statement	Description
BEGINTRANS	Starts a transaction log of all operations.
COMMITTRANS	Ends maintenance of the transaction log.
SAVEPOINT	Marks points within the transaction log to which the transaction can be rolled back.
ROLLBACK [[ALL]]	Restores the state of the database to what it was at a specified save point or at the beginning of the transaction.

Specifying a Transaction Block

Bracketing certain portions of your code with **BEGINTRANS** and **COMMITTRANS** statements provides a mechanism to retract all changes made to a database within the transaction block. No results of processing within the block will become part of the database unless everything resulting from processing in the block becomes part of the database. By following the block with a **CHECKPOINT** statement, you can guarantee that all results of the block are written immediately to disk. Save points allow you to define points within transactions to which the state of the database can be rolled back. Don't confuse a save point with the **CHECKPOINT** command. The **SAVEPOINT** function doesn't write records to disk. It simply returns integer identifiers for each of the markers it sets in the transaction log.

The Transaction Log

The **BEGINTRANS** statement causes ISAM to start logging every change made to the database. Note that ISAM only logs changes made to the database—it does not keep track of execution flow of your program. After **BEGINTRANS** is executed, changes are still made to the database, but ISAM can backtrack through those changes by referring to the transaction log. Included in the log entries are each of the save points you set with the **SAVEPOINT** function. If a **ROLLBACK** statement is executed at some point within the transaction block, ISAM checks the log and restores the database to the state it was in when the specified save point was executed. This includes removing any changes that were made to data records since the save point, and restoring all indexes to the state they were in at the save point. **BEGINTRANS** and **COMMITTRANS** take no arguments.

Using Save Points

Since ISAM maintains only one transaction log, you cannot nest one transaction within another. However, the ability to set multiple save points within a transaction supplies similar functionality with greater flexibility. While **BEGINTRANS** and **COMMITTRANS** serve as block delimiters for multiple ISAM data exchange calls, you can use save points to delimit smaller data-exchange blocks within a transaction. The **SAVEPOINT** function takes no argument, but returns an integer that identifies the save point that was set.

ROLLBACK uses two forms, as shown in the following table:

Statement form	Explanation
ROLLBACK [[*savepoint*]]	The *savepoint* is an integer identifier corresponding to a save point returned by the **SAVEPOINT** function. The effect of a **ROLLBACK** statement is the restoration of the database to the state it was at the named save point. If no *savepoint* is specified, the rollback proceeds to the next available save point.
ROLLBACK ALL	Restores the database to the state it had when the most recent **BEGINTRANS** was executed.

If your program executes a **ROLLBACK** statement outside a transaction block, specifies a non-existent save point, or executes a qualified **ROLLBACK** when there are no save points within the transaction block, a trappable error is generated. Data dictionary operations (for example, **DELETEINDEX**) cannot be rolled back, since no record of them is kept in the transaction log.

A transaction can be committed without an explicit **COMMITTRANS** being executed. For example, if there is an error in an attempt to open an ISAM table or database, a **CLOSE** statement is implicitly executed on the table or database. Any time an ISAM **CLOSE** is performed (either explicitly or implicitly), any pending transaction is committed. It is not good practice to execute table-level and database-level operations within a transaction, since errors

can commit the transaction indirectly. Even if an error doesn't occur, an implict **CLOSE** may occur, committing the transaction. For example, if you delete a table from a database, and there is no other table open within the database, an implicit **CLOSE** is performed on the database. Such a **CLOSE** causes all pending transactions (even in another open database) to be committed. You should limit the use of transactions as a programming tool for controlling record-level operations.

The following example code illustrates a transaction block abstracted from the program BOOKLOOK.BAS. The initial fragment from the module-level code intercepts the code representing what the user wants to do and calls the `EditCheck` procedure to determine whether a transaction is pending, to be begun, or to be committed.

The `EDITRECORD` case in the BOOKLOOK.BAS module-level code is the only case that uses transactions. Each time the user presses Enter after editing a field in a table, a **SAVEPOINT** statement is executed just before the record is updated. The value returned by **SAVEPOINT** is saved in an element of the array variable `Marker`, as long as the user keeps editing fields, without performing other menu operations, such as displaying or searching for a new record. **SAVEPOINT** statements are executed after each succeeding edit. When the user makes a menu choice other than Edit, the transaction is committed. Any time prior to the commitment, the user can choose to Undo edits within the transaction, either as a group (**ROLLBACK ALL**), or singly in the reverse order from which they were entered, by pressing U (Undo) or Ctrl+U (Undo All).

```
' Module-level code dealing with transactions:
    .
    .
    .
    Answer% = GetInput%(BigRec)          ' Find out what the user wants to do

    IF Answer < UNDO THEN                ' Excludes UNDOALL & INVALIDKEY too
      CALL EditCheck(PendingFlag, Answer, BigRec)
    END IF
    .
    .
    .
' If user chooses "E", let him edit a field. Assign the value returned by
' SAVEPOINT to an array element, then update the table and show the changed
' field. Trap any "duplicate value for unique index" (error 86) and handle it.
' The value returned by SAVEPOINT allows rollbacks so the user can undo edits.
  CASE EDITRECORD

    IF EditField(Argument%, BigRec, Letter$, EDITRECORD, Answer%) THEN
        ' You save a sequence of savepoint identifiers in an array so
        ' you can let the user roll the state of the file back to a
        ' specific point. The returns from SAVEPOINT aren't guaranteed
        ' to be sequential.
      n = n + 1                  ' Increment counter first so save point
      Marker(n) = SAVEPOINT      ' is synced with array-element subscript.
```

```
    Alert$ = "Setting Savepoint number " + STR$(Marker(n))
    CALL ShowMessage(Alert$, 0)
    ON ERROR GOTO MainHandler
    SELECT CASE BigRec.TableNum    ' Update the table being displayed
      CASE cBookStockTableNum
        UPDATE BigRec.TableNum, BigRec.Inventory
      CASE cCardHoldersTableNum
        UPDATE BigRec.TableNum, BigRec.Lendee
    END SELECT
    ON ERROR GOTO 0
ELSE
    COMMITTRANS                  ' Use COMMITTRANS abort transaction if
    PendingFlag = FALSE          ' the user presses ESC.
    n = 0                        ' Reset array counter.
END IF
.
.
.
' If user wants to Undo all or some of a series of uncommitted edits,
' make sure there is a pending transaction to undo, then restore the state
' of the file one step at a time, or altogether, depending on whether U or
' Ctrl+U was entered.
CASE UNDO, UNDOALL
  IF PendingFlag = TRUE THEN
    IF n < 1 THEN
      CALL ShowMessage("No pending edits left to Undo...", 0)
    ELSE
      IF Answer = UNDO THEN
        Alert$ = "Restoring back to Savepoint # " + STR$(Marker(n))
        CALL ShowMessage(Alert$, 0)
        ROLLBACK Marker(n)
        n = n - 1
      ELSE                       ' If it's not UNDO, it must be UNDOALL
        CALL ShowMessage("Undoing the whole last series of edits", 0)
        ROLLBACK ALL
        n = 0
      END IF
  END IF
ELSE
  CALL ShowMessage("There are no pending edits left to Undo...", 0)
END IF
```

```
DEFINT A-Z
'********************************* EditCheck SUB ****************************
'* The EditCheck procedure monitors what the user wants to do, and if the  *
'* choice is EDITRECORD, makes sure that a transaction is begun, or if it  *
'* already has begun, continues it. If a transaction has been pending, and *
'* the user chooses anything except EDITRECORD, then the transaction is    *
'* committed.                                                              *
'*                              Parameters:                                *
'*   Pending      A flag that indicates whether transaction is pending     *
'*   Task         Tells what operation the user wants to perform now       *
'*   TablesRec    Structure containing information about the tables        *
'***************************************************************************
SUB EditCheck (Pending, Task, TablesRec AS RecStruct)
  ' First, decide if this is a new or pending transaction, or not one at all
  ' The only transaction in this program keeps edits to the current record
  ' pending until the user moves on to a new record or a new operation
  ' (for example a Reorder).

    IF Task = EDITRECORD THEN
      IF Pending = FALSE THEN
        BEGINTRANS
        Pending = TRUE
      END IF
    ELSEIF Pending = TRUE THEN  'Equivalent to Task<>EDITRECORD AND Pending=TRUE
      COMMITTRANS
      Pending = FALSE
      CALL DrawIndexBox(TablesRec.TableNum, 0)
    END IF
END SUB
```

Maintaining Physical and Logical Data Integrity

The "physical integrity" of a database is what guarantees you will be able to use the database. ISAM maintains this physical integrity as a matter of course whenever you use the database. However, circumstances can intervene that corrupt a database. For example, power to your system could be interrupted while ISAM is in the process of actually writing data to disk. When such a "crash" occurs while a file is open, the consequences are unpredictable. For example, even if no drastic damage is done, the crash may occur before all relevant indexes in the database can be updated. Similarly, some physical mishap could corrupt the file while you are not working on the database. If someone opened the file with another program such as a word processor, and modified it, its physical integrity would be compromised. In these types of situations, you can use the ISAMREPR utility to recover the undamaged parts of the database and restore its physical integrity. Making frequent backups of database files is an important element of maintaining physical integrity. You can do this with any commercial backup program, or simply with the operating system COPY command, since all the parts of an ISAM database are contained within a single disk file.

ISAM speed depends on the fact that it writes changes to the disk periodically, rather than immediately. At the same time, something like an equipment failure could occur between the time a change is made in a table and the time the change is written to disk. In such a case, the data would be lost. This type of loss can occur when the program sits idle for a while after changes are made to a table. To minimize the danger, BASIC checks the amount of time that passes and compares that to the number of times the keyboard is polled while a program is sitting in an "idle loop." For example, when **INKEY$** or an **INPUT** statement is used to poll the keyboard for input, if a certain number of keyboard checks are made, or a certain amount of time passes without a keystroke, ISAM writes all changed buffers to disk. As soon as a keystroke occurs or buffers are flushed, the checking process starts anew.

During transactions, the changes are actually made to the file on the normal basis. Their purpose is as a programming aid, rather than a form of integrity insurance. If changes are rescinded by rollbacks before the transaction is committed, the transaction log is used to restore the database to the proper state. If an equipment failure occurs before a transaction is committed, the disk file represents the state to which the transaction had progressed, rather than the state prior to the transaction. This means that the changes cannot be rescinded by rollbacks if the failure occurs within the transaction. Conversely, when a transaction concludes, not everything is necessarily written immediately to the physical disk file. The ISAM engine performs disk writes using algorithms that give priority to performance. Therefore, there may be a lag between the time when a transaction is committed and time the final pieces of data are written to disk. As in other situations, if no keyboard input occurs within a certain period after the transaction is committed, ISAM automatically writes the state of the tables to disk. In the event of a loss of power between the end of the transaction and automatic disk write, changes not yet written to disk can be lost. This could include some part of the end of the transaction. Therefore, although this eventuality is very unlikely, ISAM internally cannot guarantee a per-transaction level of data integrity.

In a program compiled with the /D option, an implicit **CHECKPOINT** statement is performed each time a **DELETE**, **INSERT**, **UPDATE**, or ISAM **CLOSE** statement is executed. If you are very concerned with obtaining maximum data integrity, and are willing to sacrifice speed, compile your program with /D.

You can write code to enhance the logical and physical integrity of your database. The **CHECKPOINT** statement forces a physical write to disk of all data in the ISAM buffers. However, with a large database, placing **CHECKPOINT** statements in too many points in a program can significantly inhibit performance.

Record Variables as Subsets of a Table's Columns

You can open a table with a *tabletype* that is a subset of a record within the table. To associate the subset data type with the columns in the table, you specify it as the *tabletype* argument to the **OPEN** statement you use to open the table. When you actually fetch a record from the table, you specify the variable (of type *tabletype*) in a **RETRIEVE** statement. That variable must have the same type as the *tabletype* argument. If it is not the same type (even though it may be a valid subset in the sense that the element names are precisely the same, etc.), a trappable error occurs. Note however, that error checking in the QBX environment is more elaborate than in programs compiled from the command line. In a separately compiled program such an error

may not be generated. In the BOOKLOOK.BAS example discussed earlier, if you wanted to open the table BookStock, but not access values in the Publisher and Price columns, you could declare a user-defined type as follows:

```
TYPE SmallStock
   IDnum      AS DOUBLE
   Title      AS STRING * 50
   Author AS STRING * 36
END TYPE
```

The order in which the elements are specified is unimportant as long as the names are the same. You can still use indexes based on all the columns in the table, but you would not be able to transfer values to and from fields in the Publisher and Price columns.

ISAM transfers values between a table and its corresponding structured variable by name, rather than by position. Therefore, in creating a subset the order of the elements can vary, as long as the names are the same. In other words, you can subtract columns as long as you preserve the original names precisely, regardless of their position. If the data types associated with the element names do not correspond to those in the table, or if a column name is spelled differently, a trappable error occurs. Therefore, you cannot simply change the data type, or the name, of a column.

Note that if the length of one of the strings was greater than the length originally declared in BookStock, an error would be generated. Once the subset type is declared, you can open the BookStock table in BOOKS.MDB as follows:

```
OPEN "BOOKS.MDB" FOR ISAM SmallStock  "BookStock" AS #1
```

Using Multiple Files: "Relational" Databases

Because ISAM maintains everything you need for a database in the multiple tables in the single database file, you rarely need to work with other files. Since a single ISAM database file can be as large as 128 megabytes, most of what other database systems do with multiple files can be handled in a single ISAM database file. In a program that accesses multiple files, table names in different files can be identical because an ISAM table name is maintained internally as a combination of the database name plus the table name.

When you open multiple database files, the number of tables that can be opened simultaneously depends on how many database files are open, as shown in the following table:

Number of open databases	Maximum tables among databases
1	Thirteen user tables may be open simultaneously.
2	Ten tables may be open simultaneously among the 2 open databases.
3	Seven tables may be open simultaneously among the 3 open databases.
4	One per database.

ISAM Utilities

The ISAMIO, ISAMCVT, ISAMREPR, and ISAMPACK utilities are described in the following sections. You can supply these programs to users of the ISAM programs you write.

During Setup you chose whether or not ISAMIO, ISAMCVT, and ISAMPACK would run as stand-alone executable files, or require the TSR. If you want the utilities configured differently, either for yourself or your users, run Setup again. (ISAMREPR can only be configured as a stand-alone program; it cannot use the TSR). When you use the TSR with the utilities, use PROISAMD.EXE, so the data dictionary functions will be available.

ASCII Import/Export Utility (ISAMIO)

If you have an ISAM database table that you want to convert to a simple ASCII text file (or vice versa), you can use the ISAMIO utility. The syntax for ISAMIO is as follows:

ISAMIO /{I | E | H | ?} *asciifile databasename tablename specfile* {/A | /C} [[/F[[: *width*]]]]

Command-line argument	Description
/I	Specifies that an ISAM table is to be created from an ASCII text file. /I stands for "import."
/E	Specifies that an ASCII text file is to be created from an ISAM table. /E stands for "export."
/H or /?	Displays help for using the ISAMIO utility. Anything following these options in the command line is ignored.
asciifile	Names the ASCII file to be imported (/I) or exported (/E).
databasename	Names the database file into which the table should be placed (/I) or from which the data for the ASCII file should be taken (/E).
tablename	Names the table within the database file into which the records from the ASCII file should be placed (/I), or the table within the database from which the data for the ASCII file should be taken (/E).
/A	Specifies that data being imported (/I) should be appended to *tablename*. If *tablename* does not exist, an error message is displayed. If /A is not specified, ISAMIO imports the data into the named table based on the table description given in *specfile* (described later in this table). If no *specfile* is named (or found), an error message is displayed.

/C

When an ISAM table is being imported (/I) from an ASCII file, /C specifies that the table's column names should be taken from the first row of data in the ASCII file. If any of the specified column names are inconsistent with the ISAM naming convention, ISAMIO terminates and displays an error message. When an ISAM table is being exported (/E), /C specifies that the table's column names should appear in the ASCII file as the first row of data (when an ASCII file is being created from an ISAM table). If /A and /C are specified, an error message appears. If /C is not specified when a table is imported, ISAMIO interprets the first row in *asciifile* as the beginning of the data records, and looks for column names in *specfile*. If /C is not specified when a table is exported, the column names are not exported.

/F[[: *width*]]

Stipulates the data being imported (/I) is of fixed width, or that data being exported (/E) should be exported in fixed-width format (i.e., no separators appear in the data file). The size of the fixed width fields are specified in the first field of the *specfile*, if /F is specified. If you don't use /F, the fields are assumed to be comma delimited, with double quotation marks enclosing string data. When exporting fixed width, *width* specifies the width of binary fields. The default *width* is 512.

specfile

A file that specifies the data type and size (for strings, arrays, and user-defined types) for each column of a table. The file's format is as follows:

[[*fixedwidthsize*,]] [[*type* , [[*size*]] , [[*columnname*]]]]

Fields can be separated with spaces or commas. The *fixedwidthsize* may only appear if the /F option was specified. The other arguments appear only if the /A option was not specified (otherwise, it is ignored). The *type* is one of the indexable ISAM data types. In the case of arrays, user-defined types, and strings longer than 255 characters, specify *type* as binary. The *columnname* is any valid ISAM column name, but is ignored if the /C option is given. If *specfile* is not valid, a descriptive error message is displayed. Valid designations for *type* include binary, integer, long, real, and currency; you can also specify variabletext (vt), and variablestring (vs). If the type is one of the latter, the *size* field must appear. If *specfile* appears on the command line when exporting, a *specfile* suitable for importing is created.

To see an example of a *specfile*, you can export an existing table, such as one of the system tables, with a command line having the following form:

ISAMIO /E NUL *databasename* MSYSOBJECTS CON

This line sends the contents of the system table to NUL, then prints the *specfile* to the screen.

The ISAMCVT Utility

Microsoft BASIC includes the ISAMCVT.EXE utility for quick conversion of files created with other database-management systems to the ISAM format. You can use it to convert Btrieve and MS/ISAM (shipped with IBM BASIC Compiler version 2.0) files, as well as db/LIB files for use with ISAM. The ISAMCVT.EXE command-line syntax for db/LIB and MS/ISAM files is as follows:

ISAMCVT /{M | D} *filename tablename databasename*

Use the following syntax for Btrieve files:

ISAMCVT /B *filename tablename databasename specfile*

Command-line argument	Description
/D	Specifies that a db/LIB file is to be converted.
/M	Specifies that an MS/ISAM file is to be converted.
/B	Specifies that a Btrieve database is to be converted.

filename — The name of a data file to be converted.

tablename — The name of the ISAM table into which the converted records will be organized. This name must follow the ISAM naming convention.

databasename — The name of the ISAM database file into which the table will be placed.

specfile — You must supply this file when converting Btrieve and MS/ISAM files. It has the following form:

BASICtype, *size*, *columnname*

The BASIC*type* is the term used by Btrieve to identify the data type; the *size* is the length of the field in the Btrieve format. The *columnname* is any valid ISAM column name. The *size* is ignored for all types except String.

If *databasename* does not exist, it is created. The utility uses the file utilities supplied with the database package that created the file. For example, the Btrieve TSR must be loaded when conversion is attempted. If the other-product file utilities are not available to ISAMCVT, a message is displayed. To convert the indexes of the original Btrieve, MS/ISAM, or db/LIB file, run ISAMCVT on the file that contains the index and name the ISAM table and database to which the index applies.

Example entries in a Btrieve *specfile* might look as follows:

```
string 4 StringCol
integer 2 IntColumn
Long 10 LongColumn
Double 5 DoubleCol
```

In addition to Double and Single, DMBF and SMBF (for the corresponding Microsoft binary format) are also valid Btrieve column types.

When the conversion is done, you can open the tables from within a BASIC program and begin using them right away. The following table describes how the data types associated with the old file map to the **TYPE...END TYPE** variables you will be using in your BASIC program:

db/LIB	Btrieve	MS/ISAM	Microsoft BASIC's ISAM
Character	String	String	**STRING** * *n*
Logical	Logical	—	**INTEGER**
Numeric without decimal places	Integer or Long	Integer/Long	**INTEGER** or **LONG**
Numeric with decimal places	Single or Double	SINGLE and DOUBLE	**DOUBLE**

Date	—	—	**DOUBLE** with external Date/Time library functions
Memo	—	—	**STRING** * *n*

The Repair Utility

The Microsoft BASIC package includes the (ISAMREPR.EXE) utility to help recover databases that become corrupted. ISAMREPR can only restore physical integrity to a database (i.e., consistency among the tables in the database). ISAM does not pre-image changes made in a database and write them to a temporary file. Therefore, it is not possible to restore individual records entered if a crash occurs between the time the records are entered in the table and the next physical disk write. If this type of situation is a major concern, you can reduce the chance of losing such records by compiling programs with the /D option and making judicious use of **CHECKPOINT** statements in your program.

When ISAMREPR restores the database, it systematically goes through every table and index and recreates the database, using every piece of internally consistent information in the file. If anything is found that cannot be reconciled with the other information in the file, it is deleted. This restores consistency to the database. There is a chance that you will need to recreate some indexes. Similarly, it is possible that some blocks of data will never be reconciled with the rest of the database, and will therefore be lost.

The syntax for ISAMREPR.EXE is as follows:

ISAMREPR *databasename*

The *databasename* is the filename of the database you need to repair. ISAMREPR restores physical integrity to the database and prints messages to the screen whenever it takes an action that results in the loss of data. These messages describe the types of problems that were discovered and corrected. You can redirect this output to a file. The messages that may appear, with descriptions, are included in the following table:

Message	Explanation
Table *name* was truncated: data lost	During structural analysis of the table's data pages, an inconsistent page caused the table to be truncated as of the last uncorrupted page. This message is only given once for any affected table.
One or more records were deleted from table *name*	One of ISAM file's internal tables (MSysObjects, MSysIndexes, or MSysColumns) was found to have inconsistent data, or during the structural analysis of a table's data pages, an inconsistent data page was removed from the table (which resulted in the deletion of any table records on that page).
One or more long values were deleted from table *name*	"Long values" refer to strings longer than 255 characters, arrays, and user-defined types (not to the **LONG** data type). Their connections to the table were corrupted, so they were deleted. This message is only given once for any affected table.

Cannot repair *name*: Not a database file	Repair process has been aborted because the file was not recognizable as a database.
Cannot repair database *name*: Uncorrectable problems	Repair process has been aborted because the database cannot be repaired. Some common reasons include: system tables not found in the expected locations; any of the system tables is structurally inconsistent; information to reconstruct system data is unavailable; rebuilt system data is inconsistent; records describing system tables, columns, or indexes are inconsistent or missing.
Repair of *name* completed successfully	Repair process completed.

A repair may also be aborted for reasons having nothing to do with the state of the database. Messages resulting in such cases include (but are not limited to): `Disk full`, `Out of memory`, and `File not found`.

When you use the ISAMREPR utility it requires additional space within your database to accomplish its work. This adds at least 32K to the size of the database. Do not run the utility if your disk does not have this amount of space available in the current working directory. ISAMREPR deletes inconsistent records in tables, but does not compact after doing so. Compacting a database is described in the next section.

The ISAMPACK Utility

When tables or records are deleted from a database (either by your program, or the ISAMREPR utility), the size of your disk file does not change. Instead, the deleted data is marked, and ISAM begins to reuse the space in the file as you add to the database. The ISAMPACK utility performs two functions. First, if there is a total of 32K of data marked for deletion, ISAMPACK actually shrinks the disk file in increments of 32K. If there is not 32K of data marked for deletion, ISAMPACK has no effect on the size of the disk file. However, any time you run ISAMPACK, it removes records marked for deletion and then copies the database, table by table, and index by index into a database having the same name (if no *newdatabasename* is specified). The effect of compaction is improved performance, in the same way that compacting a hard disk improves performance.

As it compacts the database, ISAMPACK prints a report to the screen that lists the database's tables (including the types and maximum lengths of each of their columns), and the number of records in each table. It also lists (by table), all the database's indexes, the columns they are based on, and whether or not each one is unique. You can redirect this report to a file if you choose. The syntax for ISAMPACK.EXE is as follows:

ISAMPACK *databasename* [[*newdatabasename*]]

The *databasename* is the filename of the ISAM disk file. The *newdatabasename* is an optional alternate name for the compacted database. If no *newdatabasename* is given, the original database file is renamed with the filename extension .BAK either appended to *databasename* or replacing the original extension.

Converting Btrieve Code

If you have been using Btrieve as a database file manager, you may find that the ISAM integrated into Microsoft BASIC is a convenient and far less complicated substitute.

If you've read the preceding portions of this chapter, you probably have a good idea already of how using ISAM can clean up your file-access code. With ISAM, the interface between your program and your database consists only of the ISAM statements and the structured variables you define to transfer values between your program and database tables. Using ISAM requires no elaborate initialization, and using ISAM statements is much more direct than passing a long list of arguments to BTRV. When you use ISAM, you don't need any of the following:

- **DEF SEG**

  ISAM manages memory addressing for you.

- **OPEN** nul

  With ISAM, you only have to worry about your actual database file.

- **FIELD** statements

  ISAM lets you use real, structured variables for records.

- Operation codes

  ISAM provides easy-to-use (and understand) statements for database access and manipulation.

- Status codes

  Errors in ISAM are trapped like any other BASIC errors.

- FCB addresses and buffer lengths

  ISAM handles all DOS interactions invisibly.

- Key buffers and key numbers

  ISAM uses indexes and maintains them for you.

- Position blocks

  ISAM handles file position invisibly.

Your database files are created from within your BASIC programs, as are all tables and indexes. Although you do need to invoke a TSR before loading QBX when you want to develop database code within the environment, when you create a stand-alone version of the database program, you can have all file management support built into the executable file, so your user never has to do anything but fire up the program to work with a database.

Similarly, because the ISAM data dictionary and all your tables of data are saved within the same disk file, you don't have to worry about keeping track of multiple files. Btrieve's transaction processing is limited to enhancing data integrity. ISAM's **SAVEPOINT** and **ROLLBACK** features help insure data integrity, but even more importantly, they simplify programming in which you want to allow a user to rescind a block of data exchanges.

However, Btrieve offers the following features not yet available in ISAM:

- Support for multi-user networks

  Some versions of Btrieve support simultaneous multiple-user access to the same file, while the ISAM in BASIC does not.

- Automatic logical integrity protection

  Btrieve uses a pre-imaging system of temporary files for ensuring logical record integrity and consistency among files. ISAM guarantees only physical integrity. If your system crashes in the middle of a database operation, your ISAM file automatically maintains its internal consistency, but the one or two most recent edits to records may be lost. Since updates to records take place simultaneously in the record tables and the data dictionary, the possibility of inconsistent files is reduced greatly. The worst that can happen to a ISAM file is the loss of the latest edits to several recently modified records. You can minimize the effects of such losses by careful use of the **CHECKPOINT** statement, but unwritten records can be lost as a result of system crashes.

- Same database across different disks

  Btrieve permits you to extend a database across several different disks. While the maximum size (128 megabytes) of a single ISAM file means you will probably never have to partition a database in this way, if you have been using this Btrieve feature, you will have to redesign your database for ISAM.

- Seek in descending order

  When you use Get Lower from somewhere other than the second record in an index, Btrieve moves to the first match it makes by descending down the records in the index. ISAM does not offer an equivalent statement. If your code relies heavily on Btrieve's descending-order seeking, you can substitute combinations of **SEEKEQ** and **MOVEPREVIOUS**.

- Null key

  In Btrieve, when you designate a key as NULL, it is omitted from the index. There is no way to omit records having no value from the sorting order of a given index in Microsoft ISAM. Records with zero value in the current ISAM index simply sort as though they had the lowest value for that index. If your program relies on null keys in Btrieve, you will need to recode in ISAM to produce the same behavior.

The following table illustrates the correspondence between Btrieve operation codes and the ISAM statements and functions:

Btrieve code	Description	BASIC equivalent
0 (Open)	Makes file available for access.	**OPEN** statement makes tables accessible within the database file.
1 (Close)	Releases Btrieve file.	**CLOSE** statement closes ISAM tables (and its database file).
2 (Insert)	Inserts a new record in the file.	**INSERT** statement inserts record into ISAM table.
3 (Update)	Overwrites current record.	**UPDATE** statement.
4 (Delete)	Deletes current record.	**DELETE** statement.
5 (Get Equal)	Fetches the first record whose field value matches the specified key value.	**SEEKEQ + RETRIEVE** statements fetches the first matching record in the current index.
6 (Get Next)	Fetches the record immediately following the current record.	**MOVENEXT + RETRIEVE** statements.
7 (Get Previous)	Fetches the record immediately preceding the current record.	**MOVEPREVIOUS + RETRIEVE** statements.
8 (Get Greater)	Fetches the first record whose field value exceeds the specified key value.	**SEEKGT + RETRIEVE** statements fetch the first matching record in the current index.
9 (Get Greater or Equal)	Fetches the first record whose field value equals or exceeds the specified key value.	**SEEKGE + RETRIEVE** statements fetch the first matching record in the current index.
10 (Get Less Than)	Fetches the first record whose field value is less than the specified key value.	**SEEKGE + MOVEPREVIOUS + RETRIEVE** fetch the first matching record in the current index.

11 (Get Less Than or Equal)	Fetches the first record whose field value is less than or equals the specified key value.	**SEEKGT** + **MOVEPREVIOUS** + **RETRIEVE** fetch the first matching record in the current index.
12 (Get Lowest)	Fetches the first record.	**MOVEFIRST** + **RETRIEVE** statements.
13 (Get Highest)	Fetches the last record.	**MOVELAST** + **RETRIEVE** statements.
14 (Create)	Creates a Btrieve file.	**OPEN** statement. In BASIC the database files (and tables) are created by the **OPEN** statement, if they don't already exist.
15 (Stat)	Returns number of records in the file, plus other file characteristics.	**LOF** returns the number of records in the specified table.
16 (Extend)	Allows a file to be continuous across two drives.	No equivalent.
17 (Set Directory)	Change current directory.	**CHDRIVE, CHDIR.**
18 (Get Directory)	Returns current directory.	**CURDIR$.**
19 (Begin Transaction)	Marks start of a block of related operations.	**BEGINTRANS** statement.
20 (End Transaction)	Marks end of a block of related operations.	**COMMITTRANS** statement.
21 (Abort Transaction)	Restores file to its condition prior to the beginning of the transaction.	**ROLLBACKALL.**
22 (Get Position)	Returns position of the current record.	ISAM has no equivalent because there are no record numbers in ISAM.
23 (Get Direct)	Fetches the record with the specified record number.	ISAM has no equivalent because there are no record numbers in ISAM.
24 (Step Direct)	Fetches the record in the next physical location, regardless of the index.	ISAM has no equivalent because physical location is not a meaningful mapping in ISAM.

25 (Stop)	Unloads Btrieve record manager.	In compiled BASIC programs use of an external database manager is optional. After using ISAM within the QBX environment, you should unload the TSR with its /D option.
26 (Version)	Returns the Btrieve version number.	No equivalent.
27 - 30 **+100** **+200** (Multi-user Access)	These codes represent Btrieve multi-user access support.	No equivalent. ISAM does not support multi-user access.

Run-Time Error Messages and Codes

The following BASIC errors may occur as a result of ISAM statements:

Code	Error message	Special ISAM explanation (if any)
2	Syntax error	Some syntax errors are not detected until run time. For example, supplying too few *keyvalues* to a **SEEK***operand* statement.
5	Illegal function call	Many possible causes.
6	Overflow	Can occur when automatic coercion is performed between integer and long data entering the ISAM file.
7	Out of memory	/Ib: or /Ii set too small or too large; or after EMS has been allocated for ISAM, there is not enough left for QBX to use for text tables.
10	Duplicate definition	An attempt was made to execute a **CREATEINDEX** statement for an index that already exists in the database.
13	Type mismatch	Elements of the *recordvariable* are inconsistent with the types of the columns in the table.

16	String formula too complex	
52	Bad file mode	Attempted an ISAM operation on a non-ISAM file.
54	Bad filename or number	The specified file number does not identify an ISAM table or database file.
55	File already open	The specified table or file is already open; or you specified a non-ISAM file in a **DELETETABLE** statement.
64	Bad filename	Table name or database name exceeds legal length or contains illegal character.
67	Too many files	You have tried to open more than the maximum number of files. There are no more file handles available.
70	Permission denied	You attempted to open a file that was locked, or attempted to perform a file operation on a read-only file
73	Feature unavailable	User forgot to start the ISAM TSR before starting program, or tried to perform a data-dictionary operation using the reduced TSR (PROISAM, rather than PROISAMD) or .LIB configuration.
76	Path not found	The path was invalid; for example, a named directory did not exist.
81	Invalid name	Table or index name is too long or contains illegal characters.
82	Table not found	A table was specified that is not in the database, for example, in a **DELETETABLE** statement.

83	Index not found	The index specified by **SETINDEX** was not associated with the specified table.
84	Invalid column	The name specified for a column in a **CREATEINDEX** statement does not exist.
85	No current record	Occurs typically following an unsuccessful **SEEK***operand* statement or **MOVE***dest* to end of file or beginning of file.
86	Duplicate value for unique index	An attempt was made to create a unique index on a column that already contained duplicate values; or a user attempted to enter a duplicate value in a column for which a unique index exists.
87	Invalid operation on NULL index	For example, it is illegal to execute a **SEEK***operand* statement while the NULL index is current.
88	Database inconsistent	There is a problem in the database—run the ISAMREPR utility.

Chapter 11
Advanced String Storage

This chapter explains when and how to use far strings as well as how to manipulate far strings to accomplish the following programming tasks:

- Read and write far string data using **PEEK, POKE, BSAVE,** and **BLOAD.**
- Make pointers used for mixed-language programming.
- Maximize string storage space.

Far Strings Vs. Near Strings

In previous versions of Microsoft BASIC, all variable-length string data was stored in near memory, or what assembly language programmers call DGROUP. This is a relatively small portion of total memory (a maximum of 64K). Besides containing variable-length strings, DGROUP also contained the rest of the simple variables—integers, floating-point numbers and fixed strings, all constants, and the stack. Even when the only variables you used were variable-length strings, your maximum data capacity was limited to approximately 40K.

This version of Microsoft BASIC supports "far strings"—variable-length strings stored outside of DGROUP in multiple segments of far memory. This gives you 64K for far string processing in the main module, plus several additional 64K blocks depending on the specific program you write. And, by removing variable-length strings from DGROUP, you create more room for other simple variables as well.

The following table shows the key differences between near and far string storage.

Table 11.1 Storage Allocation with Near and Far Strings

Type of string data	Near string allocation	Far string allocation
String or string array created in main module	DGROUP	Independent 64K segment
Simple string created at procedure level	DGROUP	Second 64K segment—shared by all procedures (includes temporary strings)
Simple **COMMON** string	DGROUP	Third 64K segment—for all **COMMON** strings
String array created in procedure	DGROUP	Additional 64K segments—one per invocation

As you can see, you can have up to 192K of far string storage: 64K for module-level strings, 64K for procedure-level strings, and 64K for strings declared with the **COMMON** statement. If you are doing a recursive procedure, you can actually use additional segments as well—one for each invocation. The exact programming techniques needed to achieve these increased capacities are explained in the section "Maximizing String Storage Space" later in this chapter.

Note This chapter pertains only to variable-length strings. Fixed-length strings have not changed with this release of BASIC. For ease of reading, the word "string" is always used in this chapter to mean "variable-length string."

When to Use Far Strings

For many programming applications, using far strings is the preferred storage method. It gives you more space for variable-length strings and frees DGROUP to handle more integers, floating-pointing numbers, and fixed-length strings.

If your variable-length string requirements are limited, however, there are times when you are better off with near strings. One such case is when you need all available memory for code. Another is when you have very large arrays that are memory intensive. For these instances, using near strings frees at least 64K. (For details on how BASIC stores far strings, see the section "Data Structure and Space Allocation.")

Where you have very few strings and want to decrease code size, you can use near strings instead of the longer code for far strings.

Selecting Far Strings

When compiling from within QBX, far strings are the default. If you want variable-length strings stored in DGROUP, cancel the Strings in Far Memory selection in the Make EXE File dialog box.

When compiling programs from the command line, DGROUP storage is the default. To use far strings, add the /Fs option to the BASIC Compiler (BC) command line.

All programs running within the QBX environment use far string storage; no other option is available. All Quick libraries must be made using the /Fs option as well.

Direct Far-String Processing

For most applications, far-string programming requires no new techniques. BASIC automatically takes care of far-string memory management. However, if you are using either the **BLOAD**, **BSAVE**, **POKE**, or **PEEK** statement for direct processing of far strings, you must first set the current segment address to the address of the far string being manipulated. You can use the **SSEG** (*stringvariable$*) function to return the segment address of *stringvariable$*.

As an example of direct processing, suppose you have initialized the following string:

```
A$ = STRING$(1024,65)
```

To save it on disk using **BSAVE**, you do this:

```
DEFINT A-Z
DEF SEG = SSEG(a$)
Offset = SADD(a$)
Length = LEN(a$)
BSAVE "BIGSTG.TXT", Offset, Length
```

You must also set the current segment when using **BLOAD** with a far string.

```
DEFINT A-Z
' Intialize a string variable to the correct length
' by computing the source size.
OPEN "BIGSTG.TXT" FOR INPUT AS #1
Length = LOF(1)
A$ = STRING$(Length, 0)
' Calculate location of destination.
DEF SEG = SSEG(A$)
offset = SADD(A$)
BLOAD "BIGSTG.TXT", Offset
```

The following two examples use **DEF SEG** in conjunction with **PEEK** and **POKE** on far string data. These examples insert one string into another and are a direct-processing emulation of the **MID$** statement.

```
DEFINT A-Z
' Create a string of As and Bs.
A$ = STRING$(20, 65)
B$ = STRING$(40, 66)
' Calculate their offsets.
Offset1 = SADD(A$)
Offset2 = SADD(B$)
' Insert 10 As in the string of Bs.
' Same as MID$(B$, 11, 10) = A$.
DEF SEG = SSEG(A$)
FOR I = 11 to 20
   Temp = PEEK(Offset1)
   POKE Offset2 + I - 1, Temp
NEXT I
```

The preceding example needed only one use of the **DEF SEG** statement. That is because both strings were in the same segment. In the next example, one string is in the main-module string segment and the other is in the segment declared **COMMON**. This requires a separate **DEF SEG** statement for the source and the destination.

```
DEFINT A-Z
COMMON A$
' Create a string of As and Bs.
A$ = STRING$(20, 65)
B$ = STRING$(40, 66)
' Calculate their offsets.
Offset1 = SADD(A$)
Offset2 = SADD(B$)
' Insert 10 As in the string of Bs.
' Same as MID$(b$, 11, 10) = A$.
SourceSegment = SSEG(A$)
DestinationSegment = SSEG(B$)
FOR I = 11 to 20
   DEF SEG = SourceSegment
   Temp = PEEK(Offset1 + I - 11)
   DEF SEG = DestinationSegment
   POKE Offset2 + I - 1, Temp
NEXT I
```

Important Although shown here in examples, direct manipulation of far strings is recommended only as a last resort when standard BASIC programming techniques cannot achieve the desired result. Extreme caution is advised in these cases for the following reasons:

- BASIC moves string locations during run time. Therefore the **SSEG** and **SADD** functions need to be executed immediately before using **BLOAD**, **BSAVE**, **PEEK**, and **POKE**.
- BASIC can detect whether a string has changed length. Never use **POKE** on data beyond the last character in a far string or you will get a `String Space Corrupt` error. If this occurs in the QBX environment, QBX will terminate, and you should reboot your computer before restarting QBX.
- Far string descriptors have a different format than near string descriptors. If you attempt to locate far string data by using **PEEK** to look at the descriptor, you will not be able to find the data. If your applications pass far strings extensively or accessed the string descriptor in the past to obtain information, see Chapter 13, "Mixed-Language Programming with Far Strings," for the correct new way to do this.

Calculating Far-String Memory Space

When used with far strings, the **FRE** function provides new information to give you more program control. **FRE**(*stringexpression$*) compacts string storage space and then returns the space remaining in the segment containing *stringvexpression$*. **FRE**("*stringliteral*") also compacts string storage space, but it returns the space available for temporary string storage. Temporary string storage is used whenever a string expression is created, typically to the right or left of the equal sign or as an argument to a function or statement. Here are some examples:

```
PRINT A$ + B$
CALL StringManipulator( (A$) )
A$ = A$ + "$"
```

In certain instances, you may want to use the **FRE** function to see if there is enough string space for a given operation. Suppose you are going to load a string from a file and then combine it with another string. You can check the space requirements this way:

```
OPEN "bigstg1" FOR INPUT AS #1
' Skip I/O operation if out of space.
IF LOF(1) <= FRE(A$) THEN
   INPUT #1, A$
' Before concatenating, make two checks.
' First see if there's enough temporary storage space.
   IF FRE("") >= LEN(A$) + LEN(B$) THEN
' Second, see if A$'s segment has room to store the new variable.
      IF FRE(A$) >= LEN(B$) THEN
' There's room to do the operation, so do it.
         A$ = A$ + B$
      END IF
   END IF
END IF
```

Note

The output of the **FRE** function changes, depending on whether or not far strings are selected. The following table shows the differences.

Table 11.2 Use of FRE Function for Near and Far Strings

Function	Output with near strings	Output with far strings
FRE(*stringvariable*)	Unused space in DGROUP	Unused space in *stringvariable*'s segment
FRE(*"stringliteral"*)	Unused space in DGROUP	Unused space for temporary strings
FRE(-1)	Total unused memory space	Total unused memory space
FRE(-2)	Unused stack space	Unused stack space
FRE(-3)	Available EMS	Available EMS
FRE(any other numeric)	Unused space in DGROUP	Can't use with far strings

Using Far-String Pointers

Far strings are passed to procedures written in other languages through the use of data pointers. These pointers are obtained with the **SSEG, SADD**, and **SSEGADD** functions.

To pass separate segment and offset pointers, use **SSEG** for the segment and **SADD** for the offset. This is shown in the following example, where BASIC passes a string to MASM, which prints it on the screen:

```
DEFINT A-Z
DECLARE SUB PrintMessage(BYVAL Segment, BYVAL Offset)
' Create the message with the "$" terminator--
' the DOS print service routine requires it.
A$ = "This is a short example of a message" + "$"
' Call the MASM procedure with pointers
CALL PrintMessage(SSEG(A$), SADD(A$))
```

```
;************************ PRINT MESSAGE ***************
; This MASM procedure prints a BASIC far string on the screen.
; Define use and ordering of segments so it's compatible with BASIC.
        .model medium, basic
; Set up some stack space--necessary if making this into a quick library.
        .stack
        .code
; Define a public procedure that inputs two word
; variables.
        public printmessage
printmessage proc      uses ds, segmnt, offst
; Tell DOS print routine where the string is
        mov    ax, segmnt
        mov    ds, ax
        mov    dx, offst
; Call DOS print routine and return to BASIC
        mov    ah, 9
        int    21h
        ret
printmessage endp
        end
```

Note This example uses features of MASM version 5.1, including the **.MODEL** directive which establishes compatible naming, calling, and passing conventions. It also uses simplified segment directives which eliminate the need to separate **GROUP** and **ASSUME** directives. The new **PROC** directive is employed. It includes new arguments that specify automatically saved registers, define arguments to procedures, and set up test macros to use for the arguments. The **PROC** directive also causes the proper type of return to be generated automatically based on the chosen memory model and cleans up the stack.

In the next example, a far pointer to a string is passed to a C routine, which prints the string data. The far pointer is returned by the **SSEGADD** function. The far pointer is a double word with the segment contained in the high word and the offset contained in the low word.

```
' Declare external C procedure using correct naming
' and parameter passing conventions
DEFINT A-Z
DECLARE SUB PrintMessage CDECL (BYVAL FarString AS LONG)
' Create the message as an ASCIIZ string, as
' required by the C printf function.
A$ = "This is a short example of a message" + CHR$(0)
' Tell C to print the string addressed by the far pointer
CALL PrintMessage(SSEGADD(A$))
```

```
/********************* PRINT MESSAGE ******************
* This C routine prints a BASIC far string on the screen.
* Use standard i/o header */
#include <stdio.h>

/* Define a procedure which inputs a string  far  pointer */
void printmessage (char far *farpointer)
   {
   /* print the string addressed by the far pointer */
   printf( "%s\n", farpointer);
   }
```

Notice that near and far pointers passed to FORTRAN, C, MASM, and other languages are treated by those languages as unsigned values, whereas BASIC has no such data type (**INTEGER**, **LONG**, **SINGLE**, and **DOUBLE** data types are signed). This presents no problem as long as the pointers are assigned their values from the **SSEG**, **SADD** and **SSEGADD** functions. There can be problems, however, if pointers are assigned values directly from certain types of expressions. For example, suppose `A$` is a string that exists in segment 42000 at offset 40000. The following code passes the string to MASM:

```
DEFINT A-Z
DECLARE SUB PrintMessage(BYVAL Segment, BYVAL Offset)
Segment = SSEG(A$)
Offset = SADD(A$)
CALL PrintMessage(Segment, Offset)
```

The preceding method works correctly, but what if you directly assign the pointers with the following code?

```
Segment = 42000
Offset = 40000
```

In this case, these lines produce an `Overflow` error message because the maximum value for an integer data type is 32,767. To make direct assignments, use hexadecimal numbers:

```
' set the segment using the hex equivalent of 40000 decimal.
Segment = &H9C40
' Set the offset using the hex equivalent of 42000 decimal.
Offset = &HA410
```

Maximizing String Storage Space

For applications requiring 128K of strings, the easiest way to create this much space is to keep half in the module-level string segment and half in the string segment declared with **COMMON**. For example:

```
COMMON C$, D$
A$ = STRING$(32700,65)
B$ = STRING$(32700,66)
C$ = STRING$(32700,67)
D$ = STRING$(32700,68)
```

To get another 64K, call a procedure to create the rest of the strings. If you need to refer to procedure- and module-level strings at the same time, then share them, and do your string processing in the procedure. For example:

```
SUB BigStrings
SHARED A$, B$, C$, D$
E$ = STRING$(32700,69)
F$ = STRING$(32700,70)
' Place to do processing of A$, B$, C$, D$, E$ and F$
.
.
.
END SUB
```

One problem with using large procedure-level strings is that it uses up temporary string storage space—the place where string expressions are kept—because both occupy the same data segment. Therefore, the largest string expression plus the total procedure-level string space cannot exceed 64K. To prevent `Out of String Space` errors, use the methods described in the section "Calculating Far-String Memory Space" later in this chapter.

Another way to avoid an error is to create far string arrays while in the procedure. They get their own 64K segment:

```
SUB BigStrings
SHARED A$, B$, C$, D$
DIM E$(9), F$(9)
FOR I% = 0 to 9
E$(I%) = STRING$(3210,69 + i%)
F$(I%) = STRING$(3210,70 + i%)
NEXT I%
' Place to do processing of A$, B$, C$, D$, E$() and F$()
.
.
.
END SUB
```

The previous method can be expanded upon to fill all available memory space with strings. It works because BASIC creates a new string segment for every invocation of a procedure—in other words, each time the procedure is called during recursion. Here is the idea:

```
DEFINT A-Z
DECLARE SUB ManyStrings (n)
' Compute the # of 64K blocks available.
N = FRE(-1) \ 65536
CALL ManyStrings(N)

SUB ManyStrings(N)
DIM G$(1 to 1), H$(1 to 1)
G$(1) = STRING$(32700, 71)
H$(1) = STRING$(32700, 72)
N = N - 1
IF N > 0 THEN CALL ManyStrings(n)
END SUB
```

This creates 64K of strings for each recursion. A limitation is that the only strings that can be accessed are the ones that are dimensioned at the current level of recursion. In theory, this can be overcome by passing the strings from the previous level of recursion when the next call is made. But in reality, this makes for complex code, as demonstrated by the following:

```
' Define arrays which will be passed to each new level
' of recursion.
DECLARE SUB BigStrings (N%, S1$(), S2$(), S3$(), S4$())
DEFINT A-Z
DIM S1$(1 TO 2), S2$(1 TO 2), S3$(1 TO 2), S4$(1 TO 2)
```

```
' Compute the # of 64K blocks available in far memory.
N = FRE(-1) \ 65536
CLS
' Quit if not enough memory.
IF N < 1 THEN
            PRINT "Not enough memory for operation."
            END
END IF

' Start the recursion.
CALL BigStrings(N, S1$(), S2$(), S3$(), S4$())

SUB BigStrings (N, S1$(), S2$(), S3$(), S4$())
' Create a new array (up to 64K) for each level of recursion.
DIM A$(1 TO 2)
' Have N keep track of recursion level.
SELECT CASE N
' When at highest recursion level, process the strings.
        CASE 0
PRINT S1$(1); S1$(2); S2$(1); S2$(2); S3$(1); S3$(2); S4$(1); S4$(2)
        CASE 1
                A$(1) = "Each "
                A$(2) = "word "
                S1$(1) = A$(1)
                S1$(2) = A$(2)
        CASE 2
                A$(1) = "pair "
                A$(2) = "comes "
                S2$(1) = A$(1)
                S2$(2) = A$(2)
        CASE 3
                A$(1) = "from "
                A$(2) = "separate "
                S3$(1) = A$(1)
                S3$(2) = A$(2)
        CASE 4
                A$(1) = "recursive "
                A$(2) = "procedures."
                S4$(1) = A$(1)
                S4$(2) = A$(2)
END SELECT
```

```
' Keep going until we're out of memory.
IF N > 0 THEN
                N = N - 1
' For each recursion, pass in previously created arrays.
                CALL BigStrings(N, S1$(), S2$(), S3$(), S4$())
END IF

END SUB
```

Output

```
Each word pair comes from separate recursive procedures.
```

In this last example, the variable `N` has several important functions. In the beginning it contains the total number of 64K blocks available for strings. Thus, during execution of the **SELECT CASE** code in the **SUB** procedure, it can prevent `Out of String Space` errors. The `N` variable also keeps track of the level of recursion and ends the recursion at the appropriate time.

As you can see, once the code reaches the highest recursive level, the user can process all the strings he has created. This occurs when `CASE 0` is true. In this example, all the created strings are printed on the screen.

As stated earlier, each string array can be up to 64K. They are kept short here to make the demonstration practical.

Far Strings and Older Versions of BASIC

If you want to use far strings in modules written with previous versions of BASIC, recompile them using the /Fs option. (See the preceding section, "Selecting Far Strings," for details.) The only time you have to change any code is when your module uses the **VARSEG** or **VARPTR** function. If the **VARSEG** function is used to obtain the string data segment, replace it with the **SSEG** function. If you use the **VARPTR** function to access the string descriptor, remember that a far string descriptor has a different format than a near string descriptor. You will have to make code changes. (For suggestions on how to handle this, see Chapter 13, "Mixed-Language Programming with Far Strings.")

If you are linking new code containing far strings with older code containing near strings, you must recompile the old code using the /Fs option. Otherwise the program will return an error message `Low Level Initialization` and terminate.

Data Structure and Space Allocation

This section provides details about far strings which may prove helpful when doing mixed language programming or direct processing of far string data. Additional information can be found in Chapters 13, "Mixed-Language Programming with Far Strings" and 15, "Optimizing Program Size and Speed."

Far-string data structure consists, in part, of a 4-byte string descriptor located in DGROUP and the data located in multiple segments of far memory. The string descriptor contains information that BASIC uses to manage the data as it changes length and location during run time. The exact structure of the string descriptor is unavailable.

Whenever a far string (such as `A$`) is used in a program, the string refers to the offset address in DGROUP of the string descriptor. Thus if the string descriptor of `A$` is at offset &H2000, that is what gets pushed on the stack during the following call:

```
CALL DemoSub (A$)
```

In assembly language, the equivalent would be:

```
mov    ax, 2000H
push   ax
call   DemoSub
```

For all far string arrays, the array descriptor and all string descriptors—one for each element in the array—reside in DGROUP. All string data is in far memory.

Each string segment, besides storing the string data, contains a small amount of overhead used for string management. The overhead consists of 64 bytes plus an additional 6 bytes per string in the segment.

The exact number of 64K segments used to store string data is dependent on where the strings are created and how they are declared. This can be summarized as follows:

- All far string data declared with **COMMON** resides in a separate 64K segment.
- All other far strings created at the module-level, whether simple or in arrays, reside in a separate 64K segment.
- All string arrays created at the procedure level reside in a separate 64K segment. The arrays are local to the procedure and exist only until the procedure is exited. During recursion, arrays created at all levels exist up to and within the most deeply nested level. During the exit process, when the routine returns to a previous level, arrays used in the exited level are cleared.
- All simple strings created in any procedure reside in a single, separate 64K segment.

Note

The segment for procedure-level strings is also used for temporary strings. Temporary strings are created for all string expressions that appear anywhere in BASIC code. When using large string expressions, therefore, you may have to reduce the number of procedure-level strings to avoid running out of space in this segment. For an example of how to monitor this activity, see the section "Calculating Far-String Memory Space."

Chapter 12
Mixed-Language Programming

Mixed-language programming is the process of combining programs from two or more source languages. For example, mixed-language programming allows you to use Microsoft Macro Assembler (MASM) to enhance your BASIC programs. You can develop most of your program rapidly using BASIC, then use assembly language for routines that are executed many times and must run with utmost speed. Similarly, you can call your own Microsoft C, Pascal, and FORTRAN routines from within BASIC programs.

This chapter assumes that you know the languages you wish to combine, and that you know how to write, compile, and link multiple-module programs with these languages. When you finish this chapter, you will understand:

- General issues important in mixed-language programming.
- Calling between BASIC and other Microsoft high-level languages.
- Passing parameters in interlanguage calls.
- Differences in how BASIC, Pascal, FORTRAN, and C handle numeric and string data.
- Calling between BASIC and assembly language.

Information in this chapter assumes you are compiling your BASIC modules using the command-line compiler, using "near strings" (that is, without the /Fs option). If you do not plan to use the /Fs option, the information in this chapter should be all you need to combine modules written in Microsoft BASIC, C, Pascal, FORTRAN, and Macro Assembler.

Using "far strings" (compiling with the /Fs option) vastly increases the amount of space you can use for strings. However, because the string descriptors used for near strings and far strings are completely different, rules for mixed-language programming differ depending on which string model is used. Microsoft BASIC includes a set of mixed-language string-handling routines you can use in all your programming to assure portability among modules. For information on "far strings" and using the mixed-language string routines, see Chapters 11, "Advanced String Storage" and 13, "Mixed-Language Programming with Far Strings."

Note also that data storage differs for some kinds of data depending on whether you are using the command-line compiler or working within the QBX program-development environment. A table in the section "Special Data Types" later in this chapter summarizes these differences.

Warning

Routines intended for use in Quick libraries must use far strings. Non-BASIC Quick libraries written for QuickBASIC version 4.5 and earlier may have to be rewritten to use far strings before they can be used in the QBX environment. Also, you cannot use the /Ea option in QBX if you are using a Quick library that contains a non-BASIC routine that receives a BASIC array.

Note

These restrictions apply when creating mixed-language programs with Microsoft BASIC:

- Some combinations of languages may produce a `Symbol defined more than once` error when compiled for use with the BASIC run-time module. To avoid this, compile your program as a stand-alone executable (use the /O compiler option).
- Modules created with Microsoft QuickPascal are not compatible with any other language and cannot be linked into a mixed-language program
- You cannot link Pascal modules compiled with /Fpa with BASIC modules. Programs that include Pascal modules cannot use alternate math.

Organizing Mixed-Language Programs

The way you organize mixed-language programs depends on whether you run your program from within the QBX program-development environment, or compile from the command line using BASIC Compiler (BC). When your program calls other-language routines from within the QBX environment, the other-language routines must first be compiled and linked into a Quick library, and the Quick library must be loaded in QBX, as described in Chapter 19, "Creating and Using Quick Libraries."

If you compile and link your program from the DOS command line, your other-language routines do not have to be part of a library. However, the Microsoft Library Manager (LIB) is provided for this purpose if you find it more convenient. See Chapters 16, "Compiling with BC," 17, "About Linking and Libraries," and 18,"Using LINK and LIB," for information on compiling, linking, and managing libraries.

Note

It is especially important that other-language procedures be thoroughly debugged before being incorporated in a Quick library. The QBX tracing commands do not step into Quick-library procedures when tracing through a program, so debugging them from within the environment is not possible. For source-level debugging of mixed-language programs, compile and link the modules from the command line with the /Zi and /CO options (or using the appropriate settings in the QBX Make EXE dialog box). You can then use the Microsoft CodeView debugger to debug the mixed-language program at their source level.

Mixed-Language Programming Elements

Microsoft languages have special keywords that facilitate mixed-language programming. To use these keywords, you must understand certain fundamental issues.

After explaining the context of a mixed-language call, the following sections describe: how the languages differ and how to resolve these differences. The three fundamental mixed-language programming requirements are discussed:

- The naming convention
- The calling convention
- Parameter passing

Finally, issues relating to compiling and linking are discussed (including use of different memory models with C-language routines).

Making Mixed-Language Calls

Mixed-language programming always involves a function or procedure call. For example, a BASIC main module may need to execute a specific task that you would like to program separately. However, instead of calling a BASIC subprogram, you decide to call a C function.

Mixed-language calls necessarily involve multiple modules. Instead of compiling all of your source modules with the same compiler, you use different compilers. In the situation mentioned earlier, you could compile the main-module source file with BC, another source file (written in C) with the C compiler, and then link the two object files using LINK. Alternatively, you could compile the C function, and then link it into a Quick library and call the function from a program running within the QBX environment.

Any mixed language program that includes a BASIC module must have a BASIC main module. This is because BASIC requires that the environment be initialized in a unique way. No other language performs this initialization.

Figure 12.1 illustrates the syntax of a mixed-language call in which a BASIC main module calls a C function.

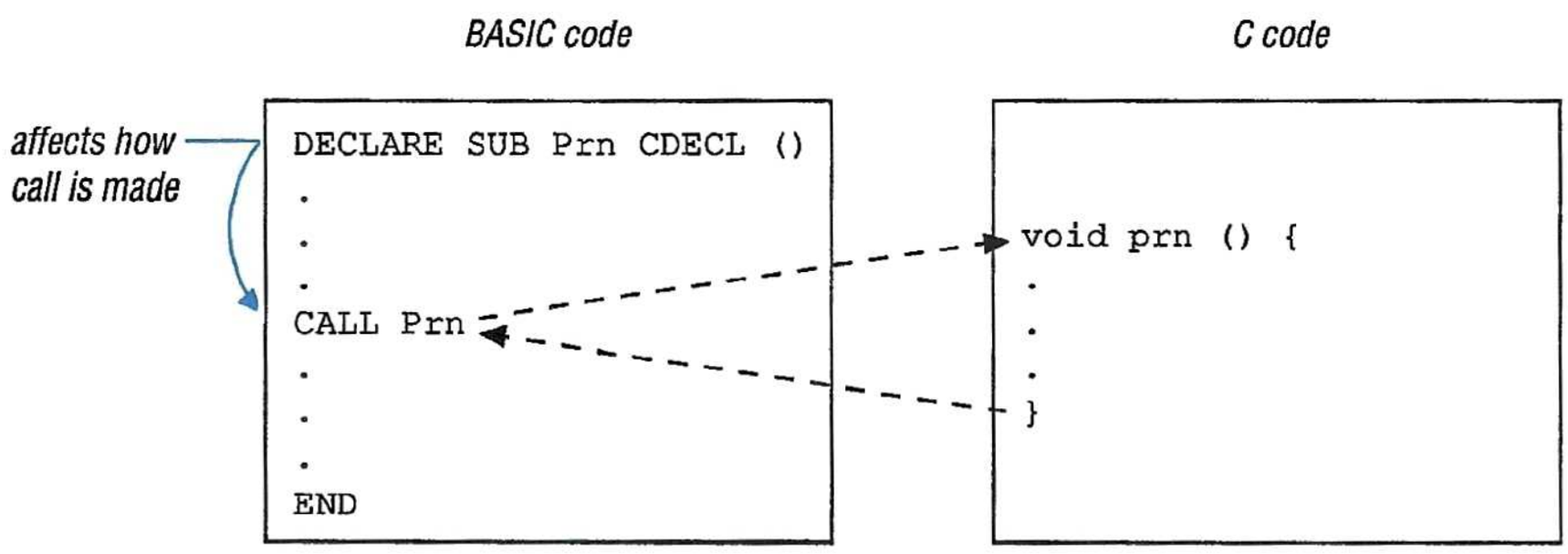


Figure 12.1 Mixed-Language Call

In Figure 12.1, the BASIC call to C is `CALL Prn`. The form is similar to a call to a BASIC **SUB** procedure. However, there are two differences between this mixed-language call and a call between two BASIC modules:

- The routine `Prn` is actually implemented in C, using standard C syntax.
- The implementation of the call in BASIC is affected by the presence of a **DECLARE** statement, which uses the **CDECL** keyword in order to create compatibility with C.

The **DECLARE** statement is an example of a mixed-language "interface" statement. Each language provides its own form of interface.

See the **DECLARE** statement in the *BASIC Language Reference* for more information.

Despite syntactic differences, BASIC **FUNCTION** and **SUB** procedures are very similar to subroutines, procedures and functions in other Microsoft languages. The principal difference is that C, Pascal, FORTRAN functions, and BASIC **FUNCTION** procedures (and assembly language procedures) can all return values, whereas the BASIC **SUB** procedure, FORTRAN **SUBROUTINE**, and Pascal procedure cannot. Table 12.1 shows the correspondence between routine calls in different languages.

Table 12.1 Language Equivalents for Routine Calls

Language	Return value	No return value
BASIC	**FUNCTION** procedure	**SUB** procedure
C	function	(void) function
Assembly	procedure	procedure
Pascal	function	procedure
FORTRAN	**FUNCTION**	**SUBROUTINE**

For example, a BASIC module can make a **SUB** procedure call to a C function declared with the **void** keyword in place of a return type. BASIC should make a **FUNCTION** procedure call in order to call a C function that returns a value; otherwise, the return value is lost.

Note

In this chapter, "routine" refers to any C function, BASIC **SUB** or **FUNCTION** procedure, or assembly language procedure that can be called from another module.

BASIC **DEF FN** functions and **GOSUB** subroutines cannot be called from another language.

Naming Convention Requirement

The calling program and the called routine must agree on the names of identifiers. Identifiers can refer to routines (functions, procedures, and subroutines) or to variables that have a public or global scope. Each language alters the names of identifiers.

"Naming convention" refers to the way a compiler alters the name of the routine (or a public variable) before placing it in an object file. Languages may alter the identifier names differently. You can choose between several naming conventions to ensure that the names in the calling routine agree with those in the called routine. If the names of public variables or called routines are stored differently in any of the object files being linked, LINK will not be able to find a match. It will instead report unresolved external references.

It is important that you adopt a compatible naming convention when you issue a mixed-language call. If the name of the called routine is stored differently in any of the object files being linked, then LINK is unable to find a match and reports an unresolved external reference.

Microsoft compilers place machine code into object files; but they also place there the names of all routines and variables that need to be accessed publicly. That way, LINK can compare the name of a routine called in one module to the name of a routine defined in another module and recognize a match. Names are stored in ASCII format.

BASIC, Pascal, and FORTRAN translate each letter to uppercase. BASIC drops its type-declaration characters (%, &, !, #, @, $). BASIC preserves the first 40 characters of any name; FORTRAN and Pascal recognize the first 31 characters.

Note

Microsoft FORTRAN prior to version 5.0 truncated identifiers to six characters. As of version 5.0, FORTRAN retains up to 31 characters of significance unless you use the /4Yt option. Microsoft Pascal prior to version 4.0 preserved only the first eight characters of a name. As of version 5.0, Pascal preserves the first 31 characters.

The C compiler does not translate any letters to uppercase, but it inserts a leading underscore (_) in front of the name of each routine. C preserves only the first 31 characters of a name.

If a name is longer than the language recognizes, additional characters are simply not placed in the object file. Also, when the mixed-language keyword **CDECL** is specified in the BASIC **DECLARE** statement, periods within a name are converted to underscores by BASIC (in addition to adding the leading underscore).

Differences in naming conventions are dealt with automatically by mixed-language keywords, as long as you follow two rules:

- If you use any FORTRAN routines that were compiled with the **$TRUNCATE** metacommand enabled or with the /4Yt command-line option, make all names six characters or less. Make all names six characters or less when using FORTRAN routines compiled with versions of the FORTRAN compiler prior to version 5.0.
- Do not use the /NOIGNORECASE (/NOI) LINK option (which causes LINK to treat identifiers in a case-sensitive manner). With C modules, this means that you must be careful not to rely upon differences between uppercase and lowercase letters when programming.

  The Microsoft C compiler drivers CL and QCL always set the /NOI option for the link stage when compiling and linking. This can be a problem if your C module contains mixed-case identifiers. For example, the C compiler translates the identifier `Name` to `_Name`, preserving the capital N. When BASIC, Pascal, and FORTRAN implement the C convention, they don't preserve case, they simply translate the characters to lowercase—so in this case, they would translate `Name` to `_name`. The identifiers `_Name` and `_name` do not match when /NOI is set. To avoid problems, make all your C-program identifiers lowercase, or link as a separate stage (i.e. use LINK, rather than CL or QCL to link), and make sure not to specify /NOI.

Note

You use the command-line option /Gc (generate Pascal-style function calls) when you compile your C modules, or if you declare a function or variable with the **pascal** keyword, the compiler will translate your identifiers to uppercase.

Figure 12.2 illustrates a complete mixed-language development example, showing how naming conventions enter into the process.

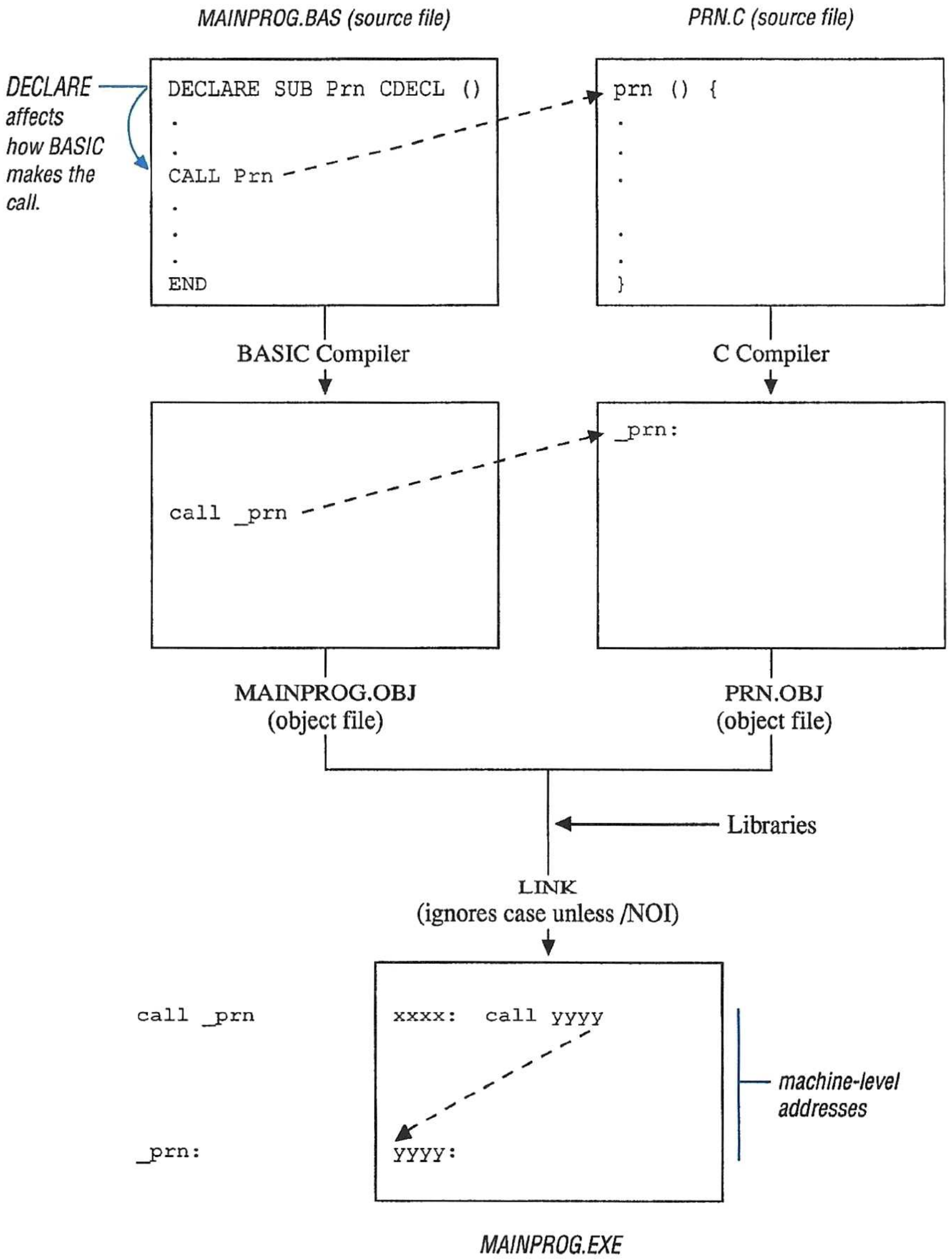


Figure 12.2 Programming with Mixed Languages

In the preceding figure, the BASIC Compiler inserts a leading underscore in front of `Prn` as it places the name into the object file, because the **CDECL** keyword directs the BASIC Compiler to use the C naming convention. BASIC will also convert all letters to lowercase when this keyword is used. (Converting letters to lowercase is not part of the C naming convention; however, it is consistent with the programming style of many C programs.)

Calling-Convention Requirement

"Calling convention" refers to the way a language implements a call. The choice of calling convention affects the actual machine instructions that a compiler generates in order to execute (and return from) a function or procedure call.

The calling convention is a low-level protocol. It is crucial that the two routines concerned (the routine issuing a call and the routine being called) recognize the same protocol. Otherwise, the processor may receive inconsistent instructions, thus causing unpredictable behavior.

The use of a calling convention affects programming in two ways:

- The calling routine uses a calling convention to determine in what order to pass arguments (parameters) to another routine. This convention can either be the default for the language, or specified in a mixed-language interface. In the following example, the **CDECL** keyword in the BASIC declaration of the C function overrides the default BASIC convention and causes the parameters to be passed in the order in which a C function normally expects to receive them:

```
DECLARE Func1 CDECL (N%, M%)
```

- The called routine uses a calling convention to determine in what order to receive the parameters passed to it. With a C function, this convention can be specified in the function definition. In the following example the **fortran** keyword in the function definition overrides the default C convention, and causes the C function to receive the parameters consistent with the default BASIC/FORTRAN/Pascal convention:

```
 int fortran func2 (int x, int y)
{
/* body of C function would go here */
}
```

In other words, the way the function is declared in BASIC determines which calling convention BASIC uses. However, in C the calling convention can be specified in the function definition. The two conventions must be compatible. It is simplest to adopt the convention of the called routine. For example, a C function would use its own convention to call another C function but must use the BASIC convention to call BASIC. This is because BASIC always uses its own convention to receive parameters. Because the BASIC and C calling conventions are different, you can change the calling convention in either the caller or the called routine, but not in both.

Effects of Calling Conventions

Calling conventions dictate three things:

- The way parameters are communicated from one routine to another (in Microsoft mixed-language programming, parameters or pointers to the parameters are passed on the stack)
- The order in which parameters are passed from one routine to another
- The part of the program responsible for adjusting the stack

Order in Which Arguments Are Pushed (BASIC, FORTRAN, Pascal)

The BASIC, FORTRAN and Pascal calling conventions push parameters onto the stack in the order in which they appear in the source code. For example, the following BASIC statement pushes argument `A` onto the stack first, then `B`, and then `C`:

```
CALL Calc( A, B, C )
```

These conventions also specify that the stack is adjusted by the called routine just before returning control to the caller. Figures 12.3 and 12.4 illustrate how the calling conventions work at the assembly language level. Note that the stack grows downward.

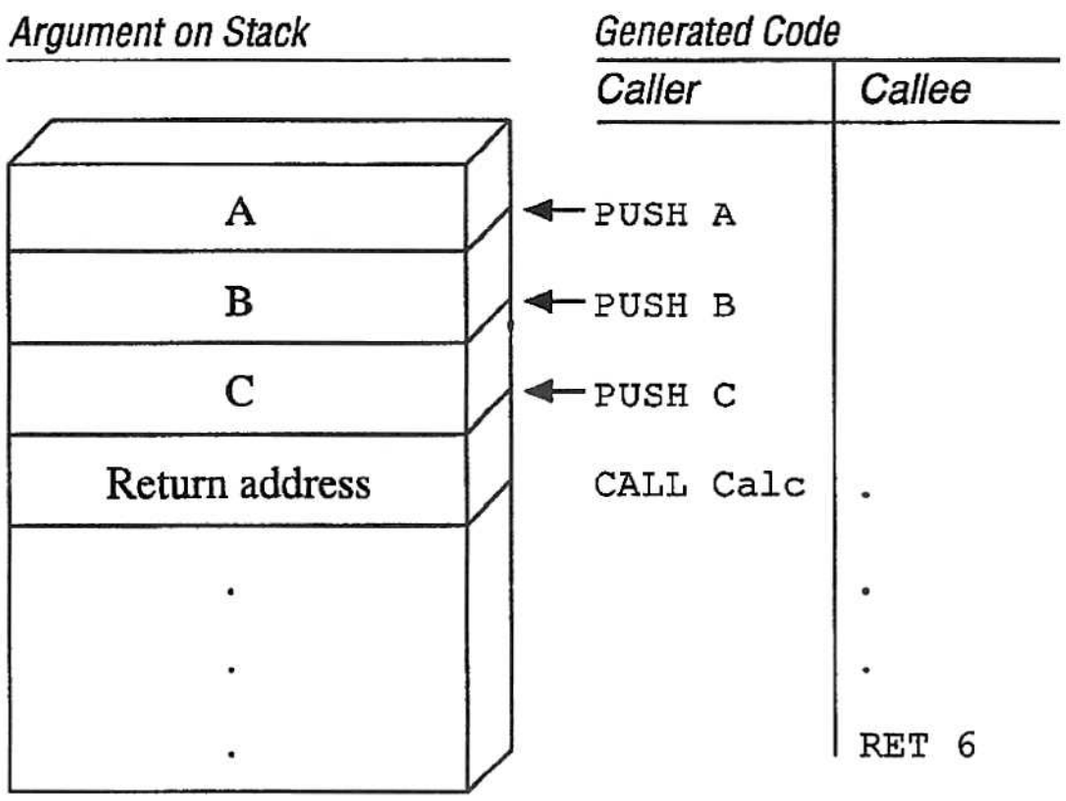


Figure 12.3 BASIC/FORTRAN/Pascal Call to Calc (A,B,C)

Order in Which Arguments Are Pushed (C)

The C calling convention pushes parameters onto the stack in the reverse order from their appearance in the source code. For example, the following C function call pushes `c` onto the stack, then `b` and finally `a`:

```
calc( a, b, c );
```

In contrast with the other high-level languages, the C calling convention specifies that a calling routine always adjusts the stack immediately after the called routine returns control.

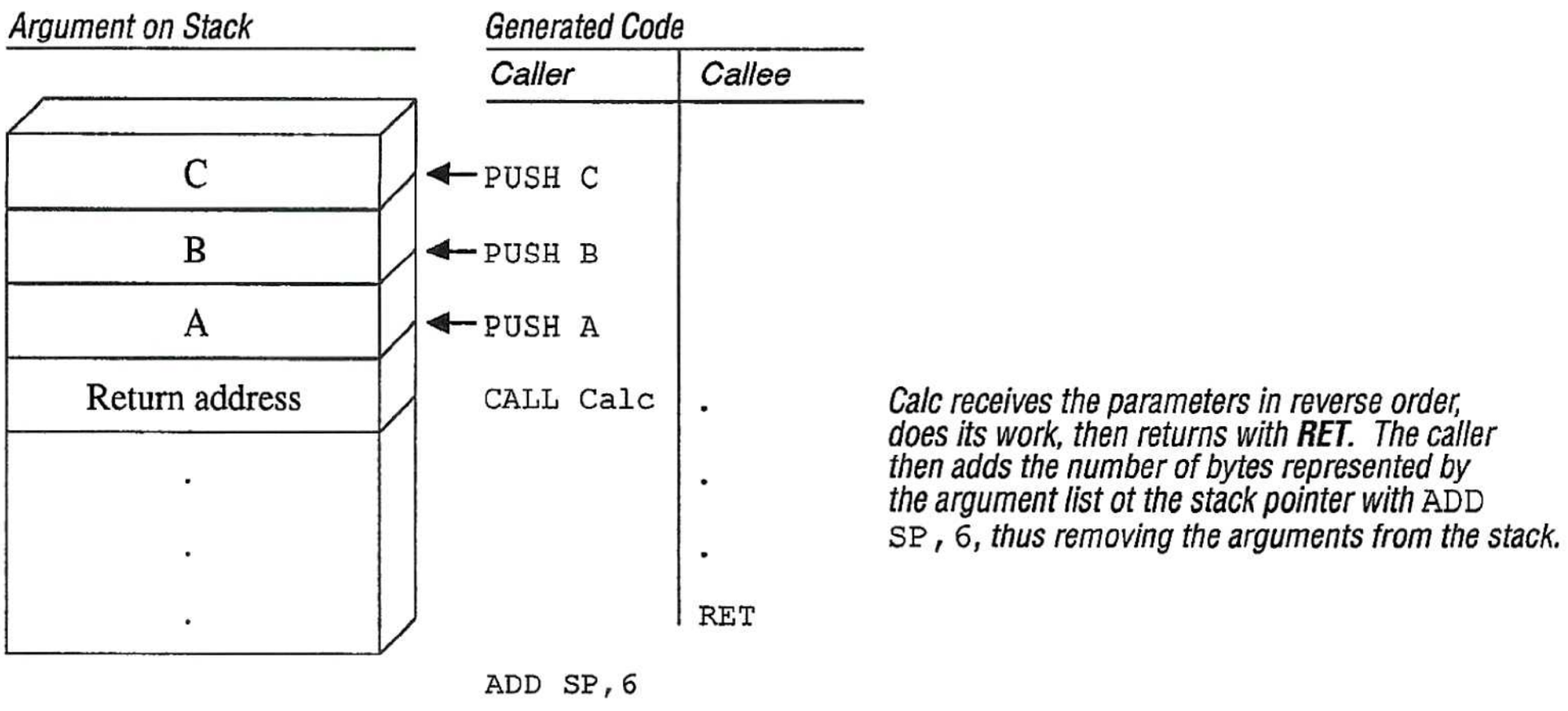


Figure 12.4 C Call to Calc (A,B,C)

The BASIC, FORTRAN, and Pascal conventions produce slightly less object code. However, the C convention makes calling with a variable number of parameters possible. (Because the first parameter is always the last one pushed, it is always on the top of the stack; therefore it has the same address relative to the frame pointer, regardless of how many parameters were actually passed.)

Note

The C-compiler **fastcall** keyword, which specifies that parameters are to be passed in registers, is incompatible with programs written in other languages. Avoid using **fastcall** or the /Gr command-line option for C functions that you intend to make public to BASIC, FORTRAN, or Pascal programs.

Parameter-Passing Requirements

The routines in program must agree on the calling convention and the naming convention; they must also agree on the method in which they pass parameters. It is important that your routines send parameters in the same way to ensure proper data transmission and correct program results.

Microsoft compilers support three methods for passing a parameter:

Method	Description
Near reference	Passes a variable's near (offset) address. This address is expressed as an offset from the beginning of the default data segment (DGROUP). This method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter changes the variable in the calling routine. BASIC passes by near reference as the default. See Chapters 11, "Advanced String Storage," and 13, "Mixed Language Programming with Far Strings," for information on passing strings stored in far memory.
Far reference	Passes a variable's far (segmented) address.This method is similar to passing by near reference, except that a longer address is passed. This method is slower than passing by near reference but is necessary when you pass data that is stored outside the default data segment. (This is an issue in BASIC or Pascal only if you have specifically requested far memory. See Table 12.5 in the section "Special Data Types," later in this chapter and Chapters 11, "Advanced String Storage," and 13, "Mixed-Language Programming with Far Strings," for information on when BASIC and QBX store data in far memory).
Value	Passes only the variable's value, not its address. With this method, the called routine knows the value of the parameter but has no access to the original variable. Changes to a value passed by a parameter have no affect on the value of the parameter in the calling routine.

These different parameter-passing methods mean that you must consider the following when programming with mixed languages:

- You need to make sure that, for each parameter, the called routine and the calling routine use the same method for passing and receiving the argument. In most cases, you will need to check the parameter-passing defaults used by each language and possibly make adjustments. Each language has keywords or language features that allow you to change parameter-passing methods.
- You may want to choose a specific parameter-passing method rather than using the defaults of any language. Table 12.2 summarizes the parameter-passing defaults for each language.

Table 12.2 Default Methods for Passing Parameters

Language	Near reference	Far reference	By value
BASIC	All		
C	Near arrays	Far arrays	All data except arrays
FORTRAN	All (medium model)	All (with large and huge models)	With Pascal or C attributes [1]
Pascal	**VAR, CONST**	**VARS, CONSTS**	Other parameters

[1] When a Pascal or C attribute is applied to a FORTRAN routine, passing by value becomes the default.

See Chapters 11, "Advanced String Storage," and 13, "Mixed-Language Programming with Far Strings," for information on how strings in far memory are passed.

Using the **BYVAL** keyword for a parameter in the BASIC declaration of an other-language routine enables you to perform a true "pass by value" to the other-language routine, as explained in the section "Using the Parameter List" later in this chapter.

Compiling and Linking

After you have written your source files and decided on a naming convention, a calling convention, and a parameter-passing convention, you are ready to compile and link individual modules.

Compiling with Correct Memory Models

With BASIC, FORTRAN, and Pascal, no special options are required to compile source files that are part of a mixed-language program. With C, not all memory models are compatible with other languages.

BASIC, FORTRAN, and Pascal use only far (segmented) code addresses. Therefore, you must use one of two techniques with C programs that call one of these languages: compile C modules in medium, large, or huge model (using the /A*x* command-line options), because these models also use far code addresses; or apply the **far** keyword to the definitions of C functions you make public. If you use the /A*x* command-line option to specify medium, large, or huge model, all your function calls become far by default. This means you don't have to declare your functions explicitly with the **far** keyword. (Note that you must also declare **extern** functions **far**, as well as public functions.)

Choice of memory model affects the default data pointer size in C and FORTRAN, although this default can be overridden with the **near** and **far** keywords. With C and FORTRAN, the choice of memory model also affects whether data items are located in the default data segment; if a data item is not located in the default data segment, it cannot be passed by near reference.

For more information about code and data address sizes in C, refer to the Microsoft C documentation.

Linking with Language Libraries

In most cases, you can easily link modules compiled with different languages. However, if any module in a program is a BASIC module, the main module of the program must be a BASIC module. When you link a program that contains a BASIC module, the BASIC main module must appear first on the LINK command line. Do any of the following to ensure that all required libraries link in the correct order:

- Put all language libraries in the same directory as the source files.
- List directories containing all needed libraries in the LIB environment variable.
- Let LINK prompt you for libraries.

In each of these cases (assuming the BASIC module appeared first on the LINK command line), LINK finds libraries in the order that it requires them. If you enter the library names on the command line, make sure you enter them in an order that allows LINK to resolve your program's external references. Here are some points to observe when specifying libraries on the command line:

- If you are listing BASIC libraries on the LINK command line, specify those libraries first.
- If you are using FORTRAN to write one of your modules, you need to link with the /NOD (no default libraries) option, and explicitly specify all the libraries you need on the LINK command line. You can also specify these libraries with an automatic-response file (or batch file), but you cannot use a default-library search.
- If your program uses FORTRAN and C, specify the library for the most recent of the two language products first. In addition, make sure that you choose a C-compatible library when you install FORTRAN.

The following example shows how to link three modules, `mod1`, `mod2`, and `mod3`, with a user library, `GRAFX`; the BASIC run-time library, `BCL70ENR.LIB`; the C run-time library, `LLIBCE`; and the FORTRAN run-time library, `LLIBFORE`:

```
LINK /NOD mod1 mod2 mod3,,,BCL70ENR+GRAFX+LLIBCE+LLIBFORE
```

Important Microsoft QuickC version 1.0 used medium model by default when you chose Compile from the Run menu and Obj from the Output Options. QuickC version 2.0 uses small model by default. Also when compiling from the command line with either the QCL or CL commands, you must specify the correct memory model. When small model is used, your C object files will not be compatible with your BASIC object files.

Linking with a C library containing graphics will result in `Duplicate definition` errors. Don't include graphics in C libraries that will be linked with BASIC.

BASIC Calls to High-Level Languages

Microsoft BASIC supports calls to routines written in Microsoft C, FORTRAN, and Pascal. This section describes the necessary syntax for calling these languages, then gives examples of each combination of BASIC with other languages. For simplicity in illustrating concepts, only integers are used as parameters in these examples. The section ends with a description of restrictions on the use of functions from the C standard library. Consult this section if the C functions called in your program use any system or memory-allocation library functions.

See the section "Handling Data in Mixed-Language Programming" later in this chapter for information on how to pass specific kinds of data.

The BASIC Interface to Other Languages

The BASIC **DECLARE** statement provides a flexible and convenient interface to other languages. It was introduced in Microsoft QuickBASIC version 4.0. Earlier versions of BASIC that do not provide the **DECLARE** statement also do not provide libraries that are compatible with other languages. The **DECLARE** statement is summarized in the following section.

The DECLARE Statement

The **DECLARE** statement's syntax differs slightly for **FUNCTION** and **SUB** procedures. For **FUNCTION** procedures, the **DECLARE** statement's syntax is as follows:

DECLARE FUNCTION *name* **CDECL ALIAS** "*aliasname*"(*parameterlist*)

For **SUB** procedures, use this syntax for the **DECLARE** statement:

DECLARE SUB *name* **CDECL ALIAS** "*aliasname*"(*parameterlist*)

The *name* argument is the name that appears in the BASIC source file for the **SUB** or **FUNCTION** procedure you wish to call. Here are the recommended steps for using the **DECLARE** statement to call other languages:

1. For each distinct interlanguage routine you plan to call, include a **DECLARE** statement at the beginning of the module-level code of any module in which the routine is called. (QBX cannot automatically generate **DECLARE** statements for other-language routines.) For example, your program may call the subprogram `Maxparam` five different times, each time with different arguments. However, you need to declare `Maxparam` just once for each module. The **DECLARE** statements must be placed near the beginning of the module, preceding all executable statements. A good way to do this is with an include file.
2. If you are calling a routine defined in a C module, use **CDECL** in the **DECLARE** statement (unless the C routine is defined with the **pascal** or **fortran** keyword). The **CDECL** keyword directs BASIC to use the C naming and calling conventions during each subsequent call to *name*.

3. If you are calling a C function with a name containing characters that would be illegal in BASIC (for example, the underscore), you can use the **ALIAS** feature, discussed in the next section. If you use the **CDECL** keyword, you can use a period in place of the underscore. BASIC then replaces the period with an underscore.
4. Use the parameter list to specify how each parameter is to be passed. See the section "Using the Parameter List" later in this chapter for information on how to use a parameter list.
5. Once the routine is properly declared, call it just as you would a BASIC **SUB** or **FUNCTION** procedure.

The other syntax elements are explained in the following sections.

Using ALIAS

As noted in the preceding section, the use of the **ALIAS** keyword may be necessary if you want to use an underscore as part of the C identifier. Similarly, though it is not likely to be a problem, C, FORTRAN, and Pascal place fewer characters of a name into an object file than BASIC (31 for all versions of C, FORTRAN version 5.0, and Pascal version 4.0, in addition to the leading underscore), which places up to 40 characters of a name into an object file.

Note You do not need the **ALIAS** feature to remove the type-declaration characters (**%**, **&**, **!**, **#**, **@**, **$**). BASIC automatically removes these characters when it generates object code. Thus, `Fact%` in BASIC matches `fact` in C.

The **ALIAS** keyword directs BASIC to place *aliasname* into the object file, instead of *name*. The BASIC source file still contains calls to *name*. However, these calls are interpreted as if they were actually calls to *aliasname*.

Example In the following example, BASIC places the *aliasname* `quad_result`, rather than the name `QuadResult`, into the object code. This avoids the use of a mixed-case identifier for the C function, but provides the same type of recognizability as the BASIC name.

```
DECLARE FUNCTION QuadResult% ALIAS "quad_result" (a, b, c)
```

Using the Parameter List

The following is the syntax for *parameterlist*. Note that you can use **BYVAL** or **SEG**, but not both:

{**BYVAL** | **SEG**} *variable* **AS** *type* ,{**BYVAL** | **SEG**} *variable* **AS** *type*...

Use the **BYVAL** keyword to declare a value parameter. In each subsequent call, the corresponding argument will be passed by value (the default method for C modules).

Note

BASIC provides two ways of "passing by value." In BASIC-only programs you can simulate passing by value by enclosing the argument in parentheses, as follows:

```
CALL Holm((A))
```

This method actually creates a temporary value, whose address is passed. The **BYVAL** keyword provides the only true method of passing by value, because the value itself is passed, not an address. Using **BYVAL** is the only way to make a BASIC program compatible with a non-BASIC routine that expects a value parameter. **BYVAL** is only for interlanguage calls; it cannot be used in calls between BASIC routines.

Use the **SEG** keyword to declare a far-reference parameter. In each subsequent call, the far (segmented) address of the corresponding argument will be passed. See the **DECLARE** statement in the *BASIC Language Reference* for information and cautions on the use of the **SEG** keyword.

You can choose any legal name for *variable*, but only the type associated with the name has any significance to BASIC. As with other variables, the type can be indicated with a type-declaration character (**%**, **&**, **!**, **#**, **@**, **$**), in an **AS** *type* clause, or by implicit declaration.

The **AS** *type* clause overrides the default type declaration of variable. The *type* field can be **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING**, **CURRENCY** or a user-defined type. Or it can be **ANY**, which directs BASIC to permit any type of data to be passed as the argument.

Examples

In the following example, `Calc2` is declared as a C routine that takes three arguments: the first two are integers passed by value, and the last is a single-precision real number passed by value.

```
DECLARE FUNCTION Calc2! CDECL (BYVAL A%, BYVAL B%, BYVAL C!)
```

The following example declares a subprogram, `Maxout`, that takes an integer passed by far reference and a double-precision real number passed by value.

```
DECLARE SUB Maxout (SEG Var1 AS INTEGER, BYVAL Var2 AS DOUBLE)
```

Alternative BASIC Interfaces

Though the **DECLARE** statement provides a particularly convenient interface, there are other methods of implementing mixed-language calls.

Instead of modifying the behavior of BASIC with **CDECL**, you can modify the behavior of C by applying the **pascal** or **fortran** keyword to the function definition. (These two keywords are functionally equivalent.) Or, you can compile the C module with the /Gc option, which specifies that all C functions, calls, and public symbols use the BASIC/FORTRAN/Pascal convention.

For example, the following C function uses the BASIC/FORTRAN/Pascal conventions to receive an integer parameter:

```
int pascal fun1(n)
int n;
{
 }
```

You can specify parameter-passing methods even though you omit the **DECLARE** statement or omit the parameter list, or both, as follows:

- You can make the call with the **CALLS** statement. The **CALLS** statement causes each parameter to be passed by far reference.
- You can use the **BYVAL** and **SEG** keywords in the argument list when you make the call.

In the following example, **BYVAL** and **SEG** have the same meaning that they have in a BASIC **DECLARE** statement. When you use **BYVAL** and **SEG** this way, however, you need to be careful because neither the type nor the number of parameters will be checked (as they would be if there were a **DECLARE** statement). Also note that, if you do not use a **DECLARE** statement, you must use either the **fortran** or **pascal** keyword in the C function definition, or compile the C function with the /Gc option.

```
CALL Fun2(BYVAL Term1, BYVAL Term2, SEG Sum);
```

Note BASIC provides a system-level function, **B_OnExit**, that can be called from other-language routines to log a termination procedure that will be called when a BASIC program terminates or is restarted when a Quick library is present. See the section "B_OnExit Routine" later in this chapter for more information.

BASIC Calls to C

This section applies the steps outlined earlier to two example programs. An analysis of programming considerations follows each example.

Calling C from BASIC with No Return Value

The following example demonstrates a BASIC main module calling a C function, `maxparam`. The function `maxparam` returns no value, but adjusts the lower of two arguments to equal the higher argument.

```
' BASIC source file - calls C function returning no value
'
' DECLARE Maxparam as subprogram, since there is no return value
' CDECL keyword causes Maxparam call to be made with C
' conventions. Integer parameters passed by near reference
' (BASIC default).
```

```
DECLARE SUB Maxparam CDECL (A AS INTEGER, B AS INTEGER)
'
X% = 5
Y% = 7
PRINT USING "X% = ## Y% = ##";X% ;Y%   ' X% and Y% before call
CALL Maxparam(X%, Y%)                  ' Call C function
PRINT USING "X% = ## Y% = ##";X% ;Y%   ' X% and Y% after call END
```

```
/* C source file */
/* Compile in MEDIUM or LARGE memory model */
/* Maxparam declared VOID because no return value */
void maxparam(p1, p2)
int near *p1;          /* Integer params received by near ref. */
int near *p2;          /* NEAR keyword not needed in MEDIUM model. */
{
   if (*p1 > *p2)
          *p2 = *p1;
   else
          *p1 = *p2;
}
```

You should keep the following programming considerations in mind when calling C from BASIC with no return value:

- Naming conventions

  The **CDECL** keyword causes `Maxparam` to be called with the C naming convention (as `_maxparam`).

- Calling conventions

  The **CDECL** keyword causes `Maxparam` to be called with the C calling convention, which pushes parameters in the reverse order to the way they appear in the source code.

- Parameter-passing methods

  Since the C function maxparam may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in C by using near pointers.

Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the C parameter declarations would use far pointers.

Calling C from BASIC with a Function Call

The following example demonstrates a BASIC main module calling a C function, `fact`. This function returns the factorial of an integer value.

```
' BASIC source file - calls C function with return value
'
' DECLARE Fact as function returning integer (%)
' CDECL keyword causes Fact% call to be made with
' C conventions. Integer parameter passed by value.

DECLARE FUNCTION Fact% CDECL (BYVAL N AS INTEGER)
'
X% = 3
Y% = 4
PRINT USING "The factorial of X% is ####"; Fact%(X%)
PRINT USING "The factorial of Y% is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END

/* C source file */
/* Compile in MEDIUM or LARGE model */
/* Factorial function, returning integer */
int fact(n)
int n; /* Integer passed by value, the C default */
{
   int result = 1;
   while (n > 0)
          result *= n--; /* Parameter n modified here */
   return(result);
}
```

You should keep the following programming considerations in mind when calling C from BASIC with a function call:

- Naming conventions

  The **CDECL** keyword causes `Fact` to be called with the C naming convention (as `_fact`).

- Calling conventions

  The **CDECL** keyword causes `Fact` to be called with the C calling convention, which pushes parameters in reverse order.

- Parameter-passing methods

  The preceding C function should receive the parameter by value. Otherwise the function will corrupt the parameter's value in the calling module. True passing by value is achieved in BASIC only by applying **BYVAL** to the parameter (in the **DECLARE** statement in this example); in C, passing by value is the default (except for arrays).

BASIC Calls to FORTRAN

This section applies the steps previously outlined to two example programs. An analysis of programming considerations follows each example.

Calling FORTRAN from BASIC—Subroutine Call

The following example demonstrates a BASIC main module calling a FORTRAN subroutine, MAXPARAM. The subroutine returns no value, but adjusts the lower of two arguments to equal the higher argument.

```
' BASIC source file - calls FORTRAN subroutine
'
DECLARE SUB Maxparam ALIAS "MAXPAR" (A AS INTEGER, B AS INTEGER)
'
' DECLARE as subprogram, since there is no return value
' ALIAS used because some FORTRAN versions recognize only the
' first 6 characters
' Integer parameters passed by near reference (BASIC default).
'
X% = 5
Y% = 7
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% before call.
CALL Maxparam(X%, Y%)                     ' Call FORTRAN function
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% after call.
END
```

```
C   FORTRAN source file, subroutine MAXPARAM
C
    SUBROUTINE MAXPARAM (I, J)
    INTEGER*2 I NEAR
    INTEGER*2 J NEAR
C
C    I and J received by near reference, because of NEAR attribute
C
    IF (I .GT. J) THEN
          J = I
    ELSE
          I = J
    ENDIF
    END
```

- Naming conventions

  By default, BASIC places all eight characters of `Maxparam` into the object file, yet some versions of FORTRAN place only the first six. This potential conflict is resolved with the **ALIAS** feature: both modules place `MAXPAR` into the object file.

- Calling conventions

  BASIC and FORTRAN use the same convention for calling.

- Parameter-passing methods

  Since the subprogram `Maxparam` may alter the value of either parameter, both arguments must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in FORTRAN by applying the **NEAR** attribute to each of the parameter declarations.

Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the **NEAR** attribute would be omitted from the FORTRAN code.

Calling FORTRAN from BASIC— Function Call

The following example demonstrates a BASIC main module calling a FORTRAN function, `FACT`. This function returns the factorial of an integer value.

```
' BASIC source file - calls FORTRAN function
'
DECLARE FUNCTION Fact% (BYVAL N AS INTEGER)
'
' DECLARE as function returning integer(%).
' Integer parameter passed by value.
'
X% = 3
Y% = 4
PRINT USING "The factorial of X%    is ####"; Fact%(X%)
PRINT USING "The factorial of Y%    is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END

C    FORTRAN source file - factorial function
C
```

```
FUNCTION FACT (N)
INTEGER*2 I
INTEGER*2 TEMP
TEMP = 1
DO 100 I = 1, N
    TEMP = TEMP * I
100       CONTINUE
FACT = TEMP
RETURN
END
```

- Naming conventions

  There are no conflicts with naming conventions because the function name, `FACT`, does not exceed the number of characters recognized by any version of FORTRAN. The type declaration character (`%`) is not placed in the object code.

- Calling conventions

  BASIC and FORTRAN use the same convention for calling.

- Parameter-passing methods

  When a parameter is passed that should not be changed, it is generally safest to pass the parameter by value. True passing by value is specified in BASIC by applying **BYVAL** to an argument in the **DECLARE** statement; in FORTRAN, the **VALUE** attribute in a parameter declaration specifies that the routine will receive a value rather than an address.

BASIC Calls to Pascal

This section applies the steps outlined previously to two example programs. An analysis of programming considerations follows each example.

Calling Pascal from BASIC—Procedure Call

The following example demonstrates a BASIC main module calling a Pascal procedure, Maxparam. Maxparam returns no value, but adjusts the lower of two arguments to equal the higher argument.

```
' BASIC source file - calls Pascal procedure
'
' DECLARE as subprogram, since there is no return value.
' Integer parameters passed by near reference (BASIC default).
'
DECLARE SUB Maxparam (A AS INTEGER, B AS INTEGER)
X% = 5
Y% = 7
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% before call.
CALL Maxparam(X%, Y%)                     ' Call Pascal function.
PRINT USING "X% = ##  Y% = ##";X% ;Y%    ' X% and Y% after call.
END
```

```
{  Pascal source code - Maxparam procedure. }

module Psub;
 procedure Maxparam(var a:integer; var b:integer);

{  Two integer parameters are received by near reference. }
{  Near reference is specified with the VAR keyword. }

 begin
    if a > b then
        b := a
    else
        a := b
 end;
end.
```

- Naming conventions

  Note that name length is not an issue because `Maxparam` does not exceed eight characters.

- Calling conventions

  BASIC and Pascal use the same calling convention.

- Parameter-passing methods

  Since the procedure `Maxparam` may alter the value of either parameter, both parameters must be passed by reference. In this case, near reference was chosen; this method is the default for BASIC (so neither **BYVAL** nor **SEG** is used) and is specified in Pascal by declaring parameters as **VAR**.

Far reference could have been specified by applying **SEG** to each argument in the **DECLARE** statement. In that case, the **VARS** keyword would be required instead of **VAR**.

Calling Pascal from BASIC—Function Call

The following example demonstrates a BASIC main module calling a Pascal function, `Fact`. This function returns the factorial of an integer value.

```
' BASIC source file - calls Pascal function
'
' DECLARE as function returning integer (%).
' Integer parameter passed by value.
'
DECLARE FUNCTION Fact% (BYVAL N AS INTEGER)
'
```

```
X% = 3
Y% = 4
PRINT USING "The factorial of X%    is ####"; Fact%(X%)
PRINT USING "The factorial of Y%    is ####"; Fact%(Y%)
PRINT USING "The factorial of X%+Y% is ####"; Fact%(X%+Y%)
END

{  Pascal source code - factorial function. }

module Pfun;
 function Fact (n : integer) : integer;

{  Integer parameters received by value, the Pascal default. }

 begin
    Fact := 1;
    while n > 0 do
        begin
            Fact := Fact * n;
            n := n - 1;           { Parameter n altered here. }
        end;
 end;
end.
```

- Naming conventions

  Note that name length is not an issue because `fact` does not exceed eight characters.

- Calling conventions

  BASIC and Pascal use the same calling convention.

- Parameter-passing methods

  The Pascal function in the preceding example should receive a parameter by value. Otherwise the function will corrupt the parameter's value in the calling module. True passing by value is achieved in BASIC only by applying **BYVAL** to the parameter; in Pascal, passing by value is the default.

Restrictions on Calls from BASIC

BASIC has a much more complex environment and initialization procedure than the other high-level languages. Interlanguage calling between BASIC and other languages is possible only because BASIC intercepts a number of library function calls from the other language and handles them in its own way. In other words, BASIC creates a host environment in which the C, Pascal and FORTRAN routines can function.

However, BASIC is limited in its ability to handle some C function calls. Also FORTRAN and Pascal sometimes perform automatic memory allocation that can cause errors that are hard to diagnose.The following sections consider three kinds of limitations: C memory-allocation functions, which may require a special declaration, implicit memory allocation performed by Pascal and FORTRAN, and a few specific C-library functions, which cannot be called at all.

Memory Allocation

If your C module is medium model and allocates memory dynamically with **malloc()**, or if you execute explicit calls to **nmalloc()** with any memory model, then you need to include the following lines in your BASIC source code before you call C:

```
DIM mallocbuf%(0 TO 2047)
COMMON SHARED /NMALLOC/ mallocbuf%()
```

The array can have any name; only the size of the array is significant. However, the name of the common block must be NMALLOC. In QBX environments, you need to put this declaration in a module that you incorporate into a Quick library. See Chapter 19, "Creating and Using Quick Libraries," for more information on Quick libraries.

The preceding example has the effect of reserving 4K of space (2 bytes * 2048) in the common block NMALLOC. When BASIC intercepts C **malloc** calls, BASIC allocates space out of this common block.

Warning

This common block is also used by FORTRAN and Pascal routines that perform dynamic memory allocation in connection with things like opening files or declaring global strings. Depending on the circumstances, however, the 4K of space may not be sufficient, and the error `Insufficient heap space` may be generated. If this happens, increase the amount of space in NMALLOC using increments of 512 or 1024.

When you make far-memory requests in mixed-language programs, you may find it useful to call the BASIC intrinsic function **SETMEM** first. This function can be used to reduce the amount of memory BASIC is using, thus freeing memory for far allocations. (An example of this use of **SETMEM** appears in the *BASIC Language Reference* and in the online Help for **SETMEM.**)

Important

When you call the BASIC **CLEAR** statement, all space allocated with near **malloc** calls is lost. If you use **CLEAR** at all, use it only before any calls to **malloc.**

Incompatible Functions

The following C functions are incompatible with BASIC and should be avoided:

- All forms of **spawn()** and **exec()**
- **system()**
- **getenv()**
- **putenv()**

Calling these functions results in the BASIC error message `Advanced feature unavailable`.

In addition, you should not link with the *x*VARSTK.OBJ modules (where *x* is a memory model) which C provides to allocate memory from the stack.

Note

The global C run-time variables `environ` and `_pgmptr` are defined as NULL. All functionality of these variables, as well as the functions noted previously, can be emulated using BASIC statements and intrinsic functions.

Allocating String Space

Other-language routines can allocate dynamic string space by calling the `GetSpace$` **FUNCTION** procedure:

```
FUNCTION GetSpace$ (x) STATIC
GetSpace$ = STRING$(x, CHR$(0))
END FUNCTION
```

The `GetSpace$` procedure returns a near pointer to a string descriptor that points to `x` bytes of string space. Because this space is managed by BASIC, it can move any time BASIC language code is executed. Therefore, the space must be accessed through the string descriptor, and the string descriptor must not be modified by other-language code. To release this space, pass the near pointer to the string descriptor to the `FreeSpace` **SUB** procedure:

```
SUB FreeSpace(a$) STATIC
A$ = ""
END SUB
```

Note that the preceding procedures deal with pointers to string descriptors, not string data. String descriptors are always in DGROUP, and therefore always accessed through near pointers. Although variable-length string data is stored in far memory within the QBX environment (or when you compile a program with /Fs) the string descriptors are still in DGROUP. For more information on far strings and mixed-language programming, see Chapters 11, "Advanced String Storage," and 13, "Mixed-Language Programming with Far Strings."

To return a string that is the result of a routine in another language, you can return a near pointer to a static string descriptor that is declared in the other-language code. A better method is to use the mixed-language string routines described in Chapter 13, "Mixed Language Programming with Far Strings." Although they are described in relation to far strings (for which they are mandatory), they are just as useful for near strings and should be used with new code. Because BASIC moves such strings around, the static string descriptor allocated by the other-language code becomes invalid after the function returns (or makes a call to any BASIC procedure).

Calling BASIC from other languages is described in the section "Calls to BASIC from Other Languages" later in this chapter. Constraints on dynamic-memory allocation in other-language routines (see the sections "Restrictions on Calls from BASIC" and "Memory Allocation" earlier in this chapter) still apply despite the use of a function like `GetSpace$`.

Performing I/O on BASIC Files

Other-language routines can perform input and output on files opened by the BASIC **OPEN** statement by calling BASIC procedures. The following example is a BASIC **SUB** that can be called to print an integer to a BASIC file opened as `Fileno%`:

```
SUB DoPrint(Fileno%, X%) STATIC
PRINT #Fileno%, X%
END SUB
```

For constraints on direct file I/O in other-language routines, see the sections "Restrictions on Calls from BASIC" and "Memory Allocation" earlier in this chapter.

Events and Errors

BASIC events including **COM**, **KEY**, **TIMER**, and **PLAY** may occur during execution of other-language code. The other-language code can allow such events to be handled by periodically calling a BASIC routine (this routine could be empty), or as follows:

- When compiling with the BC command, compile the BASIC procedure with the /V or /W option selected .
- Within the QBX environment, simply specify event-handling syntax in the procedure itself, then use the Run menu's Make EXE File or Make Library command to create an object file, or to incorporate the procedure into a Quick library.

The following BASIC **SUB** procedure lets you create BASIC errors:

```
SUB MakeError(X%) STATIC
ERROR X%
END SUB
```

When your other-language routine passes the error number to this procedure, the **ERROR** statement is executed and BASIC recovers the stack back to the previous call to non-BASIC code. The BASIC statement containing the error `ERROR X%` is the statement that **RESUME** would re-execute. **RESUME NEXT** would re-execute at the following statement. See Chapter 8, "Error Handling" for more information on using new Microsoft BASIC error-handling features.

Calling BASIC from other languages is described in the following section "Calls to BASIC from Other Languages."

Calls to BASIC from Other Languages

Microsoft C, FORTRAN, and Pascal can call routines written in Microsoft BASIC, if the main program is in BASIC. The following sections describe the necessary syntax for calling BASIC from other languages. Only simple parameter lists are used.

See the section "Handling Data in Mixed-Language Programming" later in this chapter for information on how to pass particular kinds of data.

Other Language Interfaces to BASIC

Because they share similar calling conventions, calling BASIC procedures from Pascal and FORTRAN is straightforward. With FORTRAN, you need only write an interface for each BASIC procedure that will be called, then call them as needed. When calling BASIC from Pascal, declare the BASIC routine with an **extern** procedure or function declaration (whichever is appropriate).

Remember that, although BASIC can pass data in several ways, it can only receive data that is passed by near reference. Therefore, data passed to any BASIC procedure must be passed as a near pointer. If your version of FORTRAN recognizes only the first 6 characters of a name, you should use BASIC's **ALIAS** feature if the routine you are calling has a name longer than FORTRAN can recognize.

Observe the following rules when you call BASIC from C, FORTRAN or Pascal:

- Start in a BASIC main module. You need to use the **DECLARE** statement to provide an interface to the other-language module.
- If the other language is C or Pascal , declare the BASIC routine as **extern**, and include type information for parameters. Use either the **fortran** or **pascal** keyword in the C declaration of the BASIC procedure to override the default C calling convention. If the other language is FORTRAN, use the **INTERFACE** statement to create the interfaces to BASIC routines.

- Make sure that all data is passed as near pointers. BASIC can pass data in a variety of ways, but it is unable to receive data in any form other than near reference.
  With near pointers, the program assumes that the data is in the default data segment (DGROUP). If you want to pass data that is not in the default data segment, then first copy the data to a variable that is in the default data segment (this is only a consideration with large-model C programs).
- Compile the C language modules in medium or large memory models.

Note All other-language-to-BASIC calling for programs within the QBX environment must be confined within a Quick library. In other words, a C function can call a BASIC procedure within the Quick library, but it cannot call a procedure defined within the QBX environment itself.

Calling BASIC from C

The C interface to BASIC is more complicated than the FORTRAN or Pascal interfaces. It uses standard C prototypes, with the **fortran** or **pascal** keyword. Using either of these keywords causes the routine to be called with the BASIC/FORTRAN/Pascal naming and calling conventions. The following steps are recommended for executing a mixed-language call from C:

1. Write a prototype for each mixed-language routine called. The prototype should declare the routine **extern** for the purpose of program documentation.

   Instead of using the **fortran** or **pascal** keyword, you can simply compile with the Pascal calling convention option (/Gc). The /Gc option causes all functions in the module to use the BASIC/FORTRAN/Pascal naming and calling conventions (except where you apply the **cdecl** keyword).

2. Pass near pointers to variables when calling a BASIC routine. You can obtain a pointer to a variable with the address-of (&) operator.

   In C, array names are always translated into pointers to the first element of the array; hence, arrays are always passed by reference. However, although BASIC arrays are referenced through near pointers, the pointer points to an array descriptor, not the array data itself. Therefore, other-language arrays cannot be passed directly to BASIC. The prototype you declare for your function ensures that you are passing the correct length address (that is, near or far). In BASIC the address must be near.

3. Issue a function call in your program as though you were calling a C function.

4. Always compile the C module in either medium, large, or huge model, or use the **far** keyword in your function prototype. This ensures that a far (intersegment) call is made to the routine.

There are two rules of syntax that apply when you use the **fortran** or **pascal** keyword:

- The **fortran** and **pascal** keywords modify only the item immediately to their right.
- The **near** and **far** keywords can be used with the **fortran** and **pascal** keywords in prototypes. The sequences **fortran far** and **far fortran** are equivalent.

The keywords **pascal** and **fortran** have the same effect on the program; using one or the other makes no difference except for internal program documentation. Use **fortran** to declare a FORTRAN routine, **pascal** to declare a Pascal routine, and either keyword to declare a BASIC routine.

The following example declares `func` to be a BASIC, Pascal, or FORTRAN function taking two **short** parameters and returning a **short** value.

```
extern short pascal far func( short near * sarg1, short near * sarg2 );
```

The following example declares **func** to be pointer to a BASIC, Pascal, or FORTRAN procedure that takes a **long** parameter and returns no value. The keyword **void** is appropriate when the called routine is a BASIC **SUB** procedure, Pascal procedure, or FORTRAN subroutine, since it indicates that the function returns no value.

```
extern void ( fortran far * func )( long near * larg );
```

The following example declares `func` to be a **far** BASIC **FUNCTION** procedure, Pascal function, or FORTRAN function. The routine receives a **double** parameter by reference (because it expects a pointer to a **double**) and returns a **int** value.

```
int far pascal func( near double * darg );
```

The following example is equivalent to the preceding example (**pascal far** is equivalent to **far pascal**).

```
int pascal far func( near double * darg );
```

When you call a BASIC procedure, you must use the FORTRAN/Pascal conventions to make the call. (However, if your C function calls FORTRAN or Pascal, you have a choice. You can make C adopt the conventions described in the previous section, or you can make the FORTRAN or Pascal routine adopt the C conventions.) The call must be a far call. You can insure this either by compiling in medium (or larger) model, or by using the **far** keyword, as shown in the preceding example.

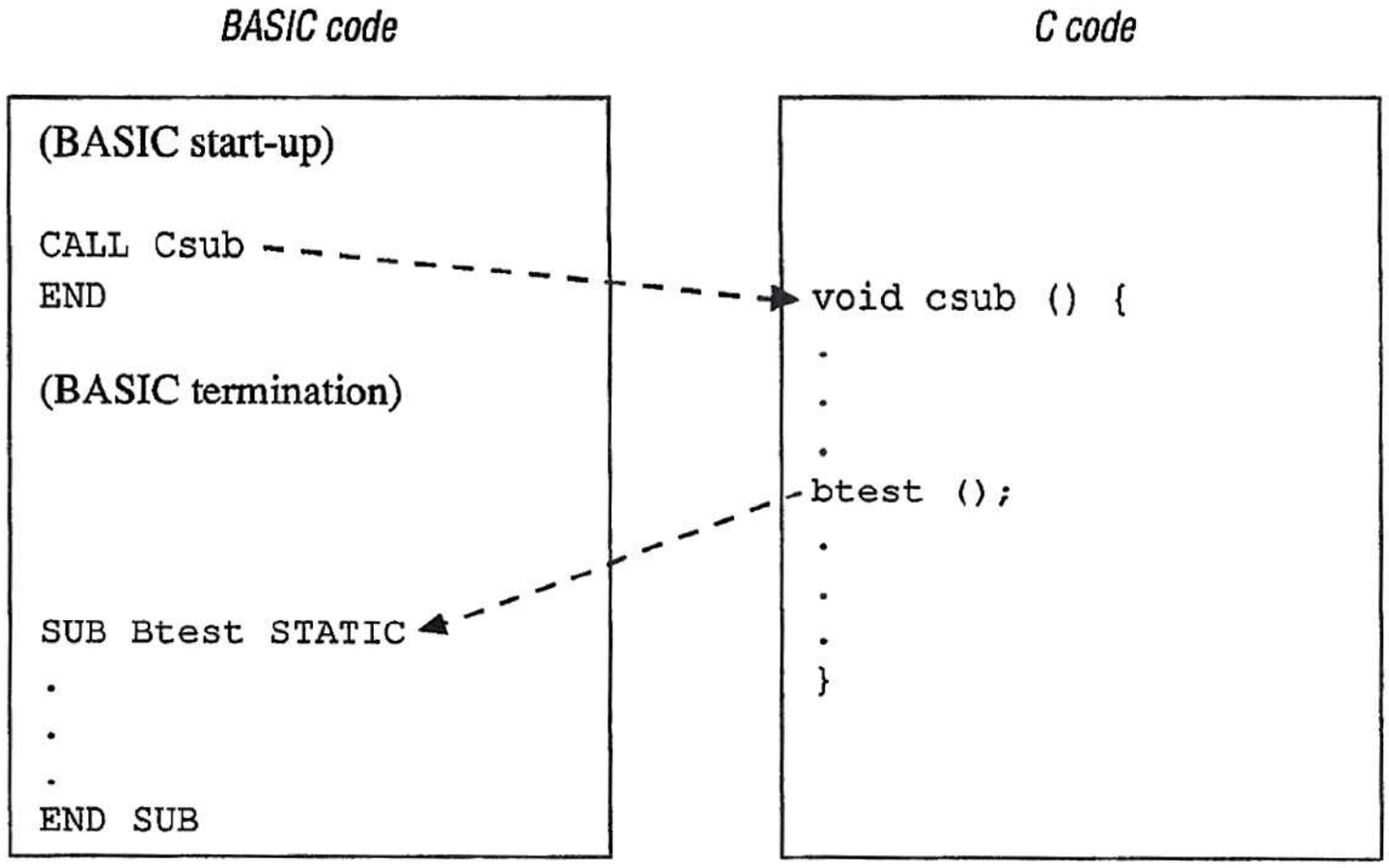


Figure 12.5 C Call to BASIC

Example The following example demonstrates a BASIC program that calls a C function. The C function then calls a BASIC function that returns twice the number passed it and a BASIC subprogram that prints two numbers.

```
' BASIC source
DEFINT A-Z
DECLARE SUB Cprog CDECL()
CALL Cprog
END
' This is the BASIC FUNCTION called in Cprog().
FUNCTION Dbl(N) STATIC
Dbl = N*2
END FUNCTION
' This is the BASIC SUB called in Cprog().
SUB Printnum(A,B) STATIC
PRINT "The first number is ";A
PRINT "The second number is ";B
END SUB
```

```
 /* C source; compile in medium or large model to insure far calls*/
extern int fortran dbl(int near *);
extern void fortran printnum(int near *, int near *);
void cprog()
{
   int near a = 5;      /* NEAR guarantees that the data */
   int near b = 6;      /* will be placed in default */
                        /* data segment (DGROUP) */
   printf("Two times 5 is %d\n", dbl(&a));
   printnum(&a, &b);
}
```

In the preceding example, note that the addresses of `a` and `b` are passed, since BASIC expects to receive addresses for parameters. Also note that the keyword **near** is used to declare each pointer in the C function declaration of `printnum`; this keyword would be unnecessary if it was known that the C module was compiled in medium model rather than large.

Calling and naming conventions are resolved by the **CDECL** keyword in BASIC's declaration of `Cprog`, and by **fortran** in C's declaration of `dbl` and `printnum`.

Calling BASIC from FORTRAN

The following example illustrates the process of calling a BASIC routine from FORTRAN. First, a call must be made from BASIC to FORTRAN, then the FORTRAN routine can call BASIC routines.

Example In this example the FORTRAN subroutine calls a BASIC **FUNCTION** that returns twice the number passed to it, then calls a BASIC **SUB** procedure that prints two numbers.

```
' BASIC source
DEFINT A-Z
DECLARE SUB Fprog ()
CALL Fprog
END
FUNCTION Dbl (N) STATIC
   Dbl = N * 2
END SUB
```

```
SUB Printnum(A,B)
   PRINT "The first number is " ; A
   PRINT "The second number is " ; B
END SUB

C          FORTRAN subroutine
C          Calls a BASIC function that receives one integer
C          and a BASIC SUB that takes two integers.
C
           INTERFACE TO INTEGER * 2 FUNCTION DBL (N)
           INTEGER * 2  N [NEAR]
           END
C
C          ALIAS attribute may necessary since BASIC recognizes more
C          than six characters of the name "Printnum" (but FORTRAN
C          version may not).
C
           INTERFACE TO SUBROUTINE PRINTN [ALIAS:  'Printn'] (N1, N2)
           INTEGER * 2 N1 [NEAR]
           INTEGER * 2 N2 [NEAR]
           END
C
C          Parameters must be declared NEAR in the parameter
C          declarations; BASIC receives only 2-byte pointers.
C
           SUBROUTINE FPROG
           INTEGER * 2 DBL
           INTEGER * 2 A, B
           A = 5
           B = 6
           WRITE (*,*) 'Two times 5 is ' , DBL(A)
           CALL PRINTN(A,B)
           END
```

In the preceding example, note that the **NEAR** attribute is used in the FORTRAN routines, so that near addresses will be passed to BASIC instead of far addresses.

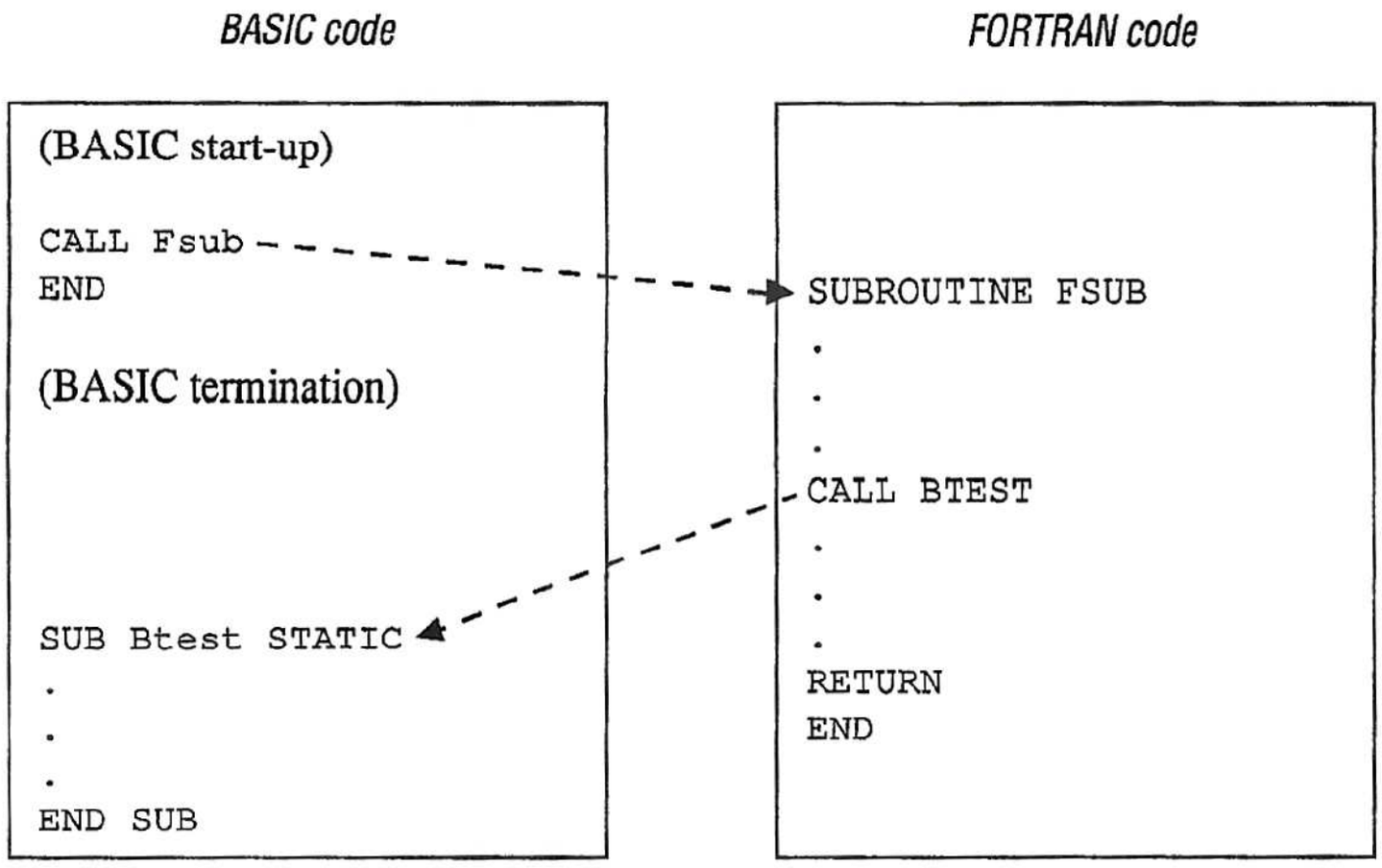


Figure 12.6 FORTRAN Call to BASIC

Calling BASIC from Pascal

The following example illustrates the process of calling a BASIC routine from Pascal. First, a call must be made from BASIC to Pascal, then the Pascal routine can call BASIC routines.

Example

In this example the Pascal procedure calls a BASIC **FUNCTION** that returns twice the number passed to it, then calls a BASIC **SUB** procedure that prints two numbers.

```
' BASIC source
DEFINT A-Z
DECLARE SUB Fprog ()
CALL Fprog
END
FUNCTION Dbl (N) STATIC
   Dbl = N * 2
END SUB

SUB Printnum(A,B)
   PRINT "The first number is " ; A
   PRINT "The second number is " ; B
END SUB

{ *       Pascal procedure    *}
{ *       Calls a BASIC function and a BASIC sub  *}

module pproc ;
procedure pprog() ;
```

```
function Dbl (var n:integer) : integer ; extern ;
procedure Printnum (var n1,n2:integer) ; extern ;
var a,b:integer ;
   begin
          a := 5
          b := 6 ;
          writeln ('Two times 5 is ' , Dbl (a) )
          Printnum(a,b)
   end
end.
```

Note that in the preceding example, every argument in the external declarations must be declared **VAR**, since BASIC can only receive near pointers as parameters.

Handling Data in Mixed-Language Programming

This section discusses naming and calling conventions in a mixed-language program. It also describes how various languages represent strings, numerical data, arrays, and logical data.

Default Naming and Calling Conventions

Each language has its own default naming and calling conventions, listed in Table 12.3

Table 12.3 Default Naming and Calling Conventions

Language	Calling convention	Naming convention	Default parameter passing method(s)
BASIC	FORTRAN/Pascal	Not case sensitive (converts to all capital letters)	Near reference
FORTRAN	FORTRAN/Pascal	Not case sensitive (converts to all capital letters)	Reference
Pascal	FORTRAN/Pascal	Not case sensitive (converts to all capital letters)	Value
C	C	Case sensitive, prepends underscore	Value (for scalar variables), reference (arrays and pointers)

BASIC Conventions

When you call BASIC routines, you must pass all arguments by near reference (near pointer).

You can modify the conventions observed by BASIC routines that call C functions by using the **DECLARE**, **BYVAL**, **SEG**, and **CALLS** keywords. For more information about the use of these keywords, see the *BASIC Language Reference*.

FORTRAN Conventions

You can modify the conventions observed by FORTRAN routines that call BASIC by using the **INTERFACE** keyword. For more information about the use of mixed-language keywords, see the *Microsoft FORTRAN Reference*.

Pascal Conventions

You can modify the conventions observed by Pascal routines that call BASIC by using the **VAR, CONST, ADR, VARS, CONSTS,** and **ADS** keywords. For more information about the use of these keywords, see the *Microsoft Pascal Compiler User's Guide*.

Passing Data by Reference or Value

The preceding sections introduced the general concepts of passing by reference and passing by value. They also noted that, by default, BASIC passes by reference, and C passes by value.

This section further describes language features that override the default. For example, using the **BYVAL** keyword in a **DECLARE** statement causes BASIC to pass a given parameter by value rather than by reference.

The next section summarizes parameter-passing methods for BASIC, discussing how to pass arguments by value, by near reference, and by far reference. Then, the same issues are discussed for C. To write a successful mixed-language interface, you must consider how each parameter is passed by the calling routine, and how each is received by the called routine.

BASIC Arguments

The default for BASIC is to pass all arguments by near reference.

Note Every BASIC **SUB** or **FUNCTION** procedure always receives data by near reference. The rest of this section summarizes how BASIC passes arguments.

Passing BASIC Arguments by Value

An argument is passed by value when the called routine is first declared with a **DECLARE** statement in which the **BYVAL** keyword is applied to the argument. Arrays and user-defined types cannot be passed by value in BASIC.

Passing BASIC Arguments by Near Reference

The BASIC default is to pass by near reference. Use of **SEG, SSEG, BYVAL,** or **CALLS** changes this default. Note that when you pass an array or a string, you are passing the descriptor (which is always in DGROUP), so even if the data is stored in far memory, the descriptor is passed by near reference.

Passing BASIC Arguments by Far Reference

Using **SEG** to modify a parameter in a preceding **DECLARE** statement causes BASIC to pass that parameter by far reference. When **CALLS** is used to invoke a routine, BASIC passes each argument in a call by far reference.

Examples

The following example passes the first argument, `A%`, by value, the second argument, `B%`, by near reference, and the third argument, `C%`, by far reference:

```
DECLARE SUB Test(BYVAL A%, B%, SEG C%)
CALL Test(X%, Y%, Z%)
```

The following example passes each argument by far reference:

```
CALLS Test2(X%, Y%, Z%)
```

C Arguments

The default for C is to pass all arrays by reference (near or far, depending on the memory model) and all other data types by value. C uses far data pointers for compact, large, and huge models, and near data pointers for small and medium models.

Passing C Arguments by Value

The C default is to pass all nonarrays (which includes all data types other than those explicitly declared as arrays) by value.

Passing C Arguments by Reference (Near or Far)

In C, passing a pointer to an data item is equivalent to passing the data item by reference. After control passes to the called function, each reference to the parameter is prefixed by an asterisk (*).

Note

To pass a pointer to a data item, prefix the parameter in the call statement with an ampersand (&). To receive a pointer to an data item, prefix the parameter's declaration with an asterisk (*). In the latter case, this may mean adding a second asterisk (*) to a parameter which already has an asterisk (*). For example, to receive a pointer by value, declare it as follows:

```
int *ptr;
```

To receive the same pointer to an integer by reference, declare it as follows:

```
int **ptr;
```

The default for arrays is to pass by reference.

Effect of Memory Models on Size of Reference

In C, near reference is the default for passing pointers in small and medium models. Far reference is the default in the compact, large, and huge models.

Near pointers can be specified with the **near** keyword, which overrides the default pointer size. However, if you are going to override the default pointer size of a parameter, then you must explicitly declare the parameter type in function prototypes as well as function definitions.

Far pointers can be specified with the **far** keyword, which overrides the default pointer size.

FORTRAN Arguments

The FORTRAN default is to pass and receive all arguments by reference. The size of the address passed depends on the memory model.

Passing FORTRAN Arguments by Value

A parameter is passed by value when declared with the **VALUE** attribute. This declaration can occur either in a FORTRAN **INTERFACE** statement (which determines how to pass a parameter) or in a function or subroutine declaration (which determines how to receive a parameter).

A function or subroutine declared with the **PASCAL** or **C** attribute will pass by value all parameters declared in its parameter list (except for parameters declared with the **REFERENCE** attribute). This change in default passing method applies to function and subroutine definitions, as well as to an **INTERFACE** statement.

Passing FORTRAN Arguments by Reference (Near or Far)

Passing by reference is the default, however, if either the **C** or **PASCAL** attribute is applied to a function or subroutine declaration, then you need to apply the **REFERENCE** attribute to any parameter of the routine that you want passed by reference.

Use of Memory Models and FORTRAN Reference Parameters

Near reference is the default for medium-model FORTRAN programs; far reference is the default for large- and huge-model programs.

Note

Versions of FORTRAN prior to 4.0 always compile in large memory model. You can apply the **NEAR** attribute to reference parameters in order to specify near reference.You can apply the **FAR** attribute to reference parameters in order to specify far reference. These keywords enable you to override the default. They have no effect when they specify the same method as the default. You may need to apply more than one attribute to a given parameter. If so, enclose both attributes in brackets, separated by commas:

```
REAL*4   X [NEAR, REFERENCE]
```

Pascal Arguments

The Pascal default is to pass all arguments by value.

Passing Pascal Arguments by Near Reference

Parameters are passed by near reference when declared as **VAR** or **CONST**.

Parameters are also passed by near reference when the **ADR** of a variable, or a pointer to a variable, is passed by value. In other words, the address of the variable is first determined. Then, this address is passed by value. (This is essentially the same method employed in C.)

Passing Pascal Arguments by Far Reference

Parameters are passed by far reference when declared as **VARS** or **CONSTS**. Parameters are also passed by far reference when the **ADS** of a variable is passed by value.

Numeric and String Data

This section discusses passing and receiving different kinds of data. Discussion includes the differences in string format and methods of passing strings between BASIC and other languages.

Integer and Real Numbers

Integer and real numbers are usually the simplest kinds of data to pass between languages. However, the type of numeric data is named differently in each language; furthermore, not all data types are available in every language, and another type may have to be substituted in some cases.

Table 12.4 shows equivalent data types in BASIC, C, FORTRAN, and Pascal.

Table 12.4 Equivalent Numeric Data Types

BASIC	C	FORTRAN	Pascal
—	**short**	**INTEGER*2**	**INTEGER2**
INTEGER (*x*%)	**int**	—	**INTEGER** (default)
—	**unsigned short**[1]	—	**WORD**
—	**unsigned**	—	—
LONG(*x*&)	**long**	**INTEGER*4**	**INTEGER4**
—	—	**INTEGER** (default)	—
—	**unsigned long**[1]	—	—
SINGLE (*x*!) (default)	**float**	**REAL*4**	**REAL4**
—	—	**REAL**	**REAL** (default)
DOUBLE (*x*#)	**double**	**REAL*8**	**REAL8**
—	—	**DOUBLE PRECISION**	—
—	**long double**	**REAL*16**	**REAL16**
—	**unsigned char**	**CHARACTER*1**[2]	**CHAR**
CURRENCY (*x*@)[3]	—	—	—

1 Types **unsigned short** and **unsigned long** are not supported by BASIC or FORTRAN. Type **unsigned long** is not supported by Pascal. A signed integral type can be substituted, but the maximum range will be less.

2 The FORTRAN type **CHARACTER*1** is not the same as **LOGICAL**.

3 The **CURRENCY** type is available only in BASIC.

The FORTRAN types **COMPLEX*8** and **COMPLEX*16** are unique to FORTRAN, but can be represented with structures. The FORTRAN types **LOGICAL*2** and **LOGICAL*4** are unique to FORTRAN.

Warning As noted in Table 12.4, C sometimes performs automatic data conversions which other languages do not perform. You can prevent C from performing such conversions by declaring a variable as the only member of a structure and then passing this structure. For example, you can pass a variable `x` of type float by first declaring the structure as follows:

```
struct {
            float x;
  } x_struct;
```

If you pass a variable of type char or float by value and do not take this precaution, then the C conversion may cause the program to fail.

Strings

Strings are stored in a variety of formats. Therefore, some transformation is frequently required to pass strings between languages. This section presents the string format(s) used in each language, and then describes methods for passing strings within specific combinations of languages. Microsoft BASIC includes several special string manipulation routines designed to simplify passing strings between modules written in different languages. They are described in Chapter 13, "Mixed-Language Programming with Far Strings."

BASIC String Format

Near strings (strings generated by the command-line compiler when you do not specify the /Fs option) are stored in BASIC as 4-byte string descriptors, as shown in Figure 12.7.

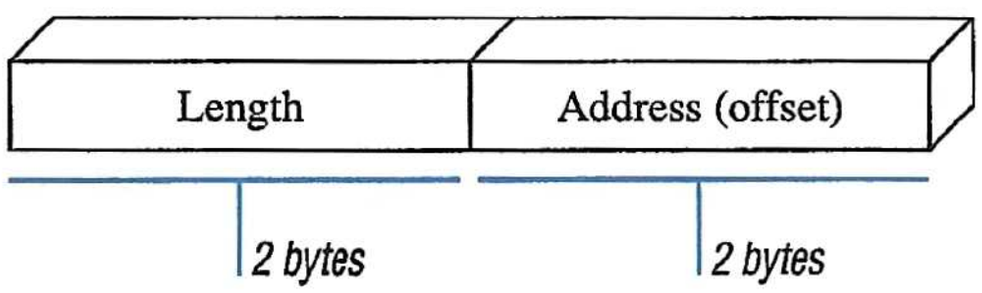


Figure 12.7 BASIC Near String-Descriptor Format

The first field of the string descriptor contains a 2-byte integer indicating the length of the actual string text. The second field contains the near address of this text. This address is an offset into the default data area (DGROUP).

Within the QBX environment the near-string model is never used. Instead, QBX (and the command-line compiler when using the /Fs option) stores the data of variable-length strings in far memory. Using far strings is described in detail in Chapters 11, "Advanced String Storage," and 13, "Mixed-Language Programming with Far Strings." Briefly, however, the difference is that with far strings, the address of the string data cannot fit into the 2 bytes normally used for the string-data address in a near-string descriptor. Instead, the DGROUP string descriptor for a far string contains "handles" (pointers representing two levels of indirection) to the information necessary to retrieve the string. Although the size and location of both types of string descriptors is the same, the information in a descriptor of a far string is totally different from the information in a string descriptor for a near string. BASIC retains the convention of having all descriptors in DGROUP, and at the same time increase the amount of available string space.

Note You cannot mix BASIC modules compiled for near strings with modules compiled for far strings. You can use BASIC modules compiled with the /Fs option in a mixed-language program if the other-language routines never try to manipulate BASIC strings by mimicking a BASIC string descriptor. If your other-language routines need to manipulate BASIC strings, you need to rewrite the source code since old methods of mimicking near-string descriptors in other languages will fail when the BASIC module uses far strings. You can guarantee portability among all modules by always using the Microsoft BASIC mixed-language string routines, and always assuming modules will be compiled with the /Fs option. If mixed-language source code is written assuming far strings, and for some reason you want to recompile the modules using the near string model, they will still work properly. In near-string code, addresses are assigned by BASIC's string-space management routines. These management routines need to be available to reassign this address whenever the length of the string changes or memory is compacted, yet these management routines are only available to BASIC. Therefore, other languages should not alter the length, or the address, of a BASIC string. See Chapter 13, "Mixed-Language Programming with Far Strings," for more information on far strings and mixed-language programming.

C String Format

C stores strings as simple arrays of bytes and uses a terminating null character (numerical 0, ASCII NUL) as a delimiter. For example, consider the string declared as follows:

```
char str[] = "String of text"
```

The string is stored in 15 bytes of memory as shown in Figure 12.8:

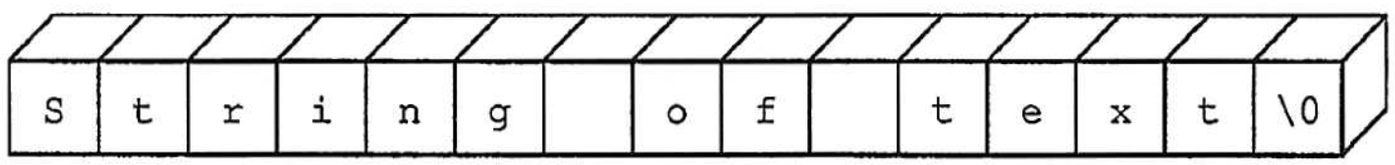


Figure 12.8 C String Format

Since `str` is an array like any other, it is passed by reference, just as other C arrays are.

FORTRAN String Format

FORTRAN stores strings as a series of bytes at a fixed location in memory. There is no delimiter at the end of the string. Consider the string declared as follows:

```
STR = "String of text"
```

The string is stored in memory as shown in Figure 12.9:

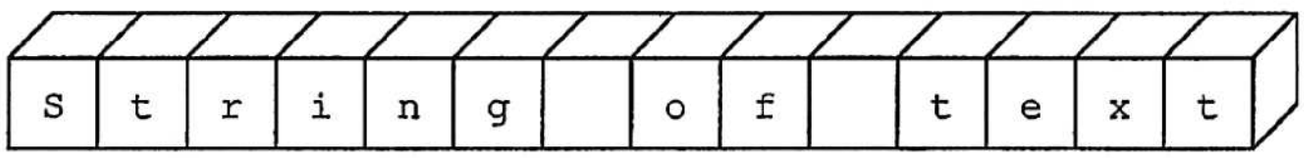


Figure 12.9 FORTRAN String Format

FORTRAN passes strings by reference, as it does all other data.

Note

Be careful using FORTRAN's variable length strings in mixed-language programming. The temporary variable used to communicate string length is not accessible to other languages. When passing such a string to another language, you need to design a method by which the target routine can find the end of the string. For example, if the target routine were in C, you could append an ASCII NUL to terminate the string before passing it.

Pascal String Format

Pascal has two types of strings, each of which uses a different format: a fixed-length type **STRING** and the variable-length type **LSTRING**. Fixed length string format is exactly like the FORTRAN format described in the preceding section. The variable-length strings are stored with the length of the string in the first byte, followed immediately by the string data itself. For example, consider the **LSTRING** declared as follows:

```
VAR STR:LSTRING(14);
STR := 'String of text'
```

The string is stored in 15 bytes of memory, as shown in Figure 12.10:

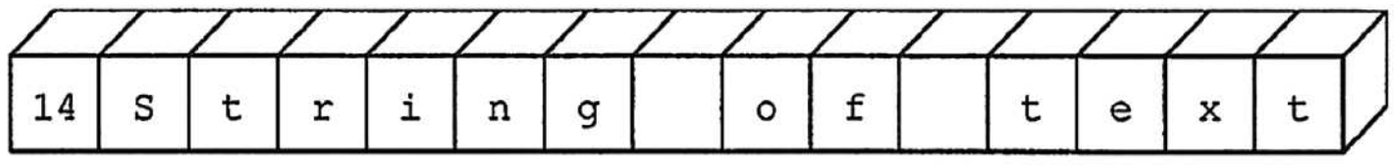


Figure 12.10 Pascal Variable-Length String Format

Passing Strings Between BASIC and Other Languages

When a BASIC string (such as `A$`) appears in an argument list, BASIC passes the address of a string descriptor rather than the actual string data. The BASIC string descriptor is not compatible with the string formats of other languages. Because no other language handles strings the way BASIC does, you cannot pass strings between BASIC and the other languages in their native forms. In previous versions of BASIC you had two choices: pass the address of the BASIC string data to the other language or mimic the form of the BASIC string descriptor in the other language, then use that to access the string as BASIC would access one of its own strings. Either of these methods worked, but greatly limited how you could work with strings in each language. You can still use either method when working with near strings in BASIC, but you must use BASIC's new string manipulation routines (described in Chapter 13, "Mixed-Language Programming with Far Strings") when working with far strings. These routines also improve the reliability and flexibility of interlanguage manipulation of near strings, and should always be used for new code. Although the old methods are described in succeeding sections, they are not recommended for new code.

Warning

When you pass a string from BASIC to another language, the called routine should under no circumstances alter the length or address of the string. Other languages lack BASIC's string-space management routines. Therefore, altering the length of a BASIC string is liable to corrupt parts of the BASIC string space. Changes that do not affect length and address, however, are safe. If the routine that receives the string calls any BASIC routine, the address of the string data may change. The second field of the string descriptor maintained by BASIC is then updated with the new address of the string text.

Passing Strings from BASIC

When you compile BASIC modules without the /Fs option, both the string descriptor and the string data are stored in DGROUP. Within the QBX environment (and when you compile a program using the /Fs compiler option), Microsoft BASIC stores variable-length string information in far memory. There is still a string descriptor in DGROUP, but it contains different information than a descriptor for a near string. Microsoft BASIC includes two new functions **SSEG** and **SSEGADD** you can use to retrieve the segment of a string, and the complete far address of far string data, respectively. You can use **SSEG** in combination with **SADD** (or just use **SSEGADD** by itself) to obtain values for directly addressing the data of strings in far memory.

Note

For far strings, **SSEGADD** performs the same role as **SADD** does for near strings. Both of these (and **SSEG**) return values. The **SEG** keyword is used differently. It is neither a statement nor a function, and is simply used to indicate that an address that is being passed is in fact a far (4-byte) address. The section "The BASIC Interface to Other Languages" earlier in this chapter describes using **SEG** in interlanguage declarations and calls.

The **SADD**, **SSEG**, **SSEGADD** and **LEN** functions extract parts of BASIC string descriptors. **SSEG** returns the segment of part of the address of the data of a variable-length string. **SADD** extracts the complete address of the data of a string stored in DGROUP (near memory), or the offset of the string data of a string stored in far memory. **SSEGADD** returns the complete (segment and offset) address of a string whose data is stored in far memory. **LEN** extracts the length of any variable-length string. The results of these functions can then be passed to other languages.

BASIC should pass the result of the **SADD SSEG**, or **SSEGADD** functions by value. Bear in mind that the string's address, not the string itself, is passed by value. This amounts to passing the string itself by reference. The BASIC module passes the string address, and the other module receives the string address. The addresses returned by **SADD** and **SSEG** are declared as type integer; the address returned by **SSEGADD** is declared as a long. These are equivalent to C's near and far pointer types, respectively.

Pass `LEN(A$)` as you would normally pass a 2-byte integer. Use values returned by **SADD**, **SSEG** or **SSEGADD** immediately because several BASIC operations may cause movement of strings in memory. Note that **SADD** cannot be used with fixed-length strings. See Appendix B, "Data Types, Constants, Variables, and Arrays," for more information.

Passing BASIC Strings to C

Before attempting to pass a BASIC string to C, you should first make the string conform to C-string format by appending a null byte on the end with an expression as follows:

```
A$ = A$ + CHR$(0)
```

You can use either of the following two methods for passing a near string from BASIC to C:

- Pass the string address and string length as separate arguments, using the **SADD** and **LEN** functions. (If you are linking to a C run-time library routine, this is the only workable method.) In the following example, `SADD(A$)` returns the near address of the string data. This address must be passed by value, since it is equivalent to a pointer (even though it is treated by BASIC as an integer). Passing by reference would attempt to pass the address of the address, rather than the address itself.

```
DECLARE SUB Test CDECL(BYVAL S%, BYVAL N%)
CALL Test(SADD(A$), LEN(A$))
.
.
.

void Test(s, n)
char _near *s;
int n;
{  /* body of function */
 }
```

C must receive a near pointer since only the near (offset) address is being passed by BASIC. Near pointers are the default pointer size in medium-model C.

- If the string is a near string, pass the string descriptor itself, with a call statement as follows:

```
CALL Test2(A$)
```

In this case, the C function must declare a structure for the parameter that has the appropriate fields (address and length) for a BASIC string descriptor. The C function should then expect to receive a pointer to a structure of this type. Such a structure declaration and function definition might look as follows:

```
struct bas_str{ /* structure declaration */
                int sd_len ;
                char near *sd_addr ;
                } ;
        test (basic_string) /* function definition */
        struct bas_str *basic_string ;
{
printf ("%S", basic_string->_sd_addr);
}
```

Note that any calls back to BASIC from within the preceding C function may invalidate the string-descriptor information. This method of mimicking BASIC string descriptors should not be used in new code. Use the BASIC string-manipulation routines described in Chapter 13, "Mixed-Language Programming with Far Strings," instead.

Passing C Strings to BASIC

When a C string appears in an argument list, C passes the address of the string. (A C string is just an array and so is passed by reference.) C can still pass near string data to BASIC in the form of a string descriptor. However, when the BASIC program uses far strings (/Fs option when compiling from command line, or by default in QBX), the special string manipulation routines described in Chapter 13, "Mixed-Language Programming with Far Strings," must be used.

To use the string descriptor method (for near strings only). first allocate a string in C; then, create a structure that mimics to a BASIC string descriptor. Pass this structure by near reference, as in the following example:

```
char cstr[] = "ABC";
struct {
          int sd_len;
          char *sd_addr;
} str_des;
str_des.sd_len = strlen(cstr);
str_des.sd_addr = cstr;
bsub(&str_des);
```

As noted previously, this method still works for BASIC programs that do not use the far strings feature of Microsoft BASIC. If you've successfully used this method with old code, it will still work. However, the Microsoft BASIC routines discussed in Chapter 13, "Mixed-Language Programming with Far Strings," can be used for both near and far strings, and make passing strings between language modules simpler and more reliable. In all cases, make sure that the string originates in C, not in BASIC. Strings originating in BASIC are subject to being moved around in memory during BASIC string management.

Passing BASIC Strings to FORTRAN

FORTRAN'S variable-length strings are unique and cannot be a part of a mixed-language interface. Use **SADD** to pass the address of the BASIC string. The FORTRAN routine should declare a character variable of the same length (which is fixed).

```
DECLARE SUB Test(BYVAL S%)     ' or use S# if the address is far
A$ = "abcd"
CALL Test (SADD(A$))           ' or SSEGADD if string is far
.
.
.
C          FORTRAN SOURCE
C
           SUBROUTINE    TEST(STRINGA)
           CHARACTER*4 STRINGA [NEAR]
```

In the preceding example, `SADD(A$)` must be passed by value, since it is actually an address, not an integer. Note that the FORTRAN declaration `CHARACTER*4 STRINGA [NEAR]` declares a fixed-length parameter received by near reference. See Chapter 13, "Mixed-Language Programming with Far Strings," for information on passing BASIC far strings.

Passing FORTRAN Strings to BASIC

FORTRAN cannot directly pass strings to BASIC because BASIC expects to receive a string descriptor when passed a string. Yet there is an indirect method for passing FORTRAN strings to BASIC, if the BASIC program does not use far strings. First, allocate a fixed-length string in FORTRAN, declare an array of two 2-byte integers, and treat the array as a string descriptor. Next, assign the length of the string to the first element, and assign the address of the string to the second element (using the **LOC** function). Finally, pass the integer array itself by reference. BASIC can receive and process this array just as it would a string descriptor

If you've successfully used this method with old code, it will still work with near-string programs. However, the Microsoft BASIC routines discussed in Chapter 13, "Mixed-Language Programming with Far Strings," can be used for near and far strings, and make passing strings between language modules simpler and more reliable. In all cases, make sure that the string originates in FORTRAN, not in BASIC. Strings originating in BASIC are subject to being moved around in memory during BASIC string management.

Passing BASIC Strings to Pascal

The same technique used for passing strings to FORTRAN can be used to pass a BASIC string to Pascal when the BASIC program uses near strings. However, the Pascal routine should declare the string as a **VAR** parameter in order to receive the string by near reference. See Chapter 13, "Mixed-Language Programming with Far Strings." The Pascal code must declare the fixed-length type `string (4)` in a separate statement, then use the declared type in a **procedure** declaration as show by the following example:

```
DECLARE SUB Test(BYVAL S%)     ' or use S& if the address is far
A$ = "abcd"
CALL Test(SADD(A$))            ' or use SSEGADD if string is far
type stype4=string(4);
procedure Test (VAR StringA:stype4);{near string}
```

Passing Pascal Strings to BASIC

To pass a Pascal string to near-string BASIC program, you can first allocate a string in Pascal. Next, create a record identical to a BASIC string descriptor. Initialize this record with the string length and address, and then pass the record by near reference. If the BASIC program uses far strings, use the string manipulation routines described in Chapter 13, "Mixed-Language Programming with Far Strings."

If you've successfully used this method with old code, it will still work with near-string programs. However, the Microsoft BASIC routines discussed in Chapter 13, "Mixed-Language Programming with Far Strings," can be used for both near and far strings, and make passing strings between language modules simpler and more reliable. In all cases, make sure that the string originates in Pascal, not in BASIC. Strings originating in BASIC are subject to being moved around in memory during BASIC string management.

Special Data Types

This section considers special types of data that are either arrays or structured types (that is, data types that contain more than one field).

Arrays

When you program in only one language, arrays do not present special problems; the language is consistent in its handling of arrays. When you program with more than one language, however, you need to be aware of two special problems that may arise with arrays:

- Arrays are implemented differently in BASIC than in other Microsoft languages, so that you must take special precautions when you pass an array from BASIC to another language (including assembly language). A reference to an array in BASIC is really a reference to an array descriptor. Array descriptors are always in DGROUP. Array data however, is sometimes stored in DGROUP and sometimes in far memory. Further, array-data storage differs slightly in QBX from array-data storage in the command-line compiler. Table 12.5 summarizes array-data storage for both QBX and the command-line compiler.

Remember, string arrays are arrays of string descriptors, not arrays of strings themselves. No matter where the actual strings are stored, the array descriptor is always in DGROUP. The array of the string descriptors referred to by the array descriptor is in DGROUP as well. With arrays of near strings (the default in a program compiled from the command line), the strings themselves are also always in DGROUP, but with far strings the actual strings are in far memory. Since QBX always uses far strings, the actual strings are always in far memory in QBX.

- Arrays are declared and indexed differently in each language.

Table 12.5 Comparison of Array Data Storage in BC and QBX

Type of array	Command-line compiler	QBX environment
Static arrays in **COMMON**	DGROUP	Far memory (unless a quick library with a corresponding **COMMON** is loaded)[1]
Other static arrays (including static arrays of variable-length strings)	DGROUP (array descriptors and string descriptors)	Far memory[1]
Dynamic arrays of variable-length strings	DGROUP (array descriptors and string descriptors)	Far memory[1]
Dynamic arrays of non-string data	Far memory	Far memory[1]

1 Arrays are stored in EMS if QBX is invoked with the /Ea option.

Passing Arrays from BASIC

Most Microsoft languages permit you to reference arrays directly. In C, for example, an array name is equivalent to the address of the first element.

This simple implementation is possible because the location of data for an array never changes.

BASIC uses an array descriptor, however, which is similar in some respects to a BASIC string descriptor, but far more complicated. The array descriptor is necessary because BASIC may shift the location of array data in memory, as BASIC handles memory allocation for arrays dynamically. See Appendix B, "Data Types, Constants, Variables, and Arrays," for specific information and cautions about passing BASIC arrays.

C, FORTRAN, and Pascal have no equivalent of the BASIC array descriptor. More importantly, they lack access to BASIC's space-management routines for arrays. Therefore, you may safely pass arrays from BASIC only if you follow three rules:

- Pass the array's address by applying the **VARPTR** function to the first element of the array, then pass the result by value. To pass the address of data stored in far memory, get the segment with **VARSEG**, the offset with **VARPTR**, then pass both by value. The target language receives the address of the first element, and considers it to be the address of the entire array. It can then access the array with its normal array-indexing syntax. The example following this list illustrates this approach.

Alternatively, only a far reference to an array can be passed by passing its first element by far reference as in the following fragments:

```
CALLS Proc1 (A(0,0))

CALL Proc2 (SEG A(0,0))
```

- The routine that receives the array must not, under any circumstances, make a call back to BASIC. If it does, then the location of the array data may change, and the address that was passed to the routine becomes meaningless.
- BASIC may pass any member of an array by value. With this method, the preceding precautions do not apply.

Example The following example passes an array from BASIC to FORTRAN:

```
' BASIC source file
DEFINT A-Z
DIM A( 1 TO 20)
DECLARE SUB ArrFix(BYVAL Addr AS INTEGER)
.
.
.
CALL ArrFix (VARPTR (A(1)) )
PRINT A(1)
END


C         FORTRAN SOURCE FILE
C
          SUBROUTINE ARRFIX (ARR)
          INTEGER * 2  ARR  [NEAR]   (20)
          ARR(1) = 5
          END
```

In the preceding example, assuming the program is compiled from the command line, BASIC considers that the argument passed is the near address of an array element. FORTRAN considers it to be the near address of the array itself. Both assumptions are correct. You can use essentially the same method for passing BASIC arrays to Pascal or C. The parameter was declared **BYVAL** and **AS INTEGER** because a near (2-byte) address needed to be passed. To pass a far address, you could use the following code instead:

```
DECLARE SUB ArrFix(BYVAL SegAdd AS INTEGER, BYVAL Addr AS INTEGER)
CALL ArrFix (VARSEG( A(0) ), VARPTR( A(0) )  )
```

The first field is the segment returned by **VARSEG**. If you use **CDECL** then be sure to pass the offset address before the segment address, because **CDECL** causes parameters to be passed in reverse order:

```
DECLARE SUB ArrFix(BYVAL Addr AS INTEGER BYVAL,SegAdd AS INTEGER)
CALL ArrFix ( VARPTR( A(0) ) , VARSEG( A(0) ) )
```

Note

You can apply the **LBOUND** and **UBOUND** functions to a BASIC array, to determine lower and upper bounds, and then pass the results to another routine. This way, the size of the array does not need to be determined in advance. See the *BASIC Language Reference* for more information on the **LBOUND** and **UBOUND** functions.

Array Indexing and Declaration

Each language varies somewhat in the way that arrays are declared and indexed. Array indexing is a source-level consideration and involves no transformation of data. There are three differences in the way elements are indexed by each language:

- The way an array's lower bound is specified differs among Microsoft languages. By default, FORTRAN indexes the first element of an array as 1. BASIC and C index it as 0. Pascal lets you begin indexing at any integer value. Recent versions of BASIC and FORTRAN also give you the option of specifying lower bounds at any integer value.
- The way the number of elements in an array is declared is different in BASIC than for the other languages. For example, in BASIC, the constants that are used in the array declaration `DIM Arr%(5,5)` represent the upper bounds of the array. Therefore, the last element is indexed as `Arr%(5,5)`. The constants used in a C array declaration represent the actual number of elements in each dimension, not upper bounds as they do in BASIC. Therefore, the last element in the C array declared as `int arr[5][5]` is indexed as `arr[4][4]`, rather than as `arr[5][5]`.
- Some languages vary subscripts in row-major order; others vary subscripts in column-major order. This issue only affects arrays with more than one dimension. When you traverse an array in row-major order, you access each column of a row before moving on to the next row. Therefore, with row-major order (the only choice with C and Pascal) each element of the rightmost subscript is accessed before the leftmost subscript changes. When you traverse an array in column-major order, you access each row of a column before moving on to the next column. Thus, with column-major order (used by BASIC by default, and the only choice in FORTRAN), each element of the leftmost subscript is accessed before the rightmost subscript changes. Thus, in C the first four elements of an array declared as `X[3][3]`, are:

```
X[0][0] X[0][1] X[0][2] X[1][0]
```

In BASIC, the corresponding four elements are:

```
X(0,0) X(1,0) X(2,0) X(0,1)
```

Similarly, the following references both refer to the same place in memory for an array:

```
arr1[2][8]    /* in C */
Arr1(8,2)     ' in BASIC
```

The preceding examples assume that the C and BASIC arrays use lower bounds of 0.

Declaring Arrays

Table 12.6 shows equivalences for array declarations in each language. In this table, *i* represents the number of elements in the leftmost dimension for column-major arrays and *I* is used for the leftmost dimension of row-major arrays. Similarly, *J* represents the number of elements in the rightmost dimension for column-major arrays and *j* is used for row-major arrays.

Table 12.6 Equivalent Array Declarations

Language	Array declaration	Notes
BASIC	**DIM** *x* (*i*–1, *J*–1)	Default is column-major storage, with lower bounds of 0. Values for *i* and *J* represent the highest subscript in the dimension, rather than the number of elements.
C	*type* [*I*][*j*] **struct** {*type* *x*[*I*][*j*];} *x*	When passed by reference. When passed by value. *I* and *j* represent the number of elements in each dimension. Always row-major storage.
FORTRAN	*type x*(*i*, *J*)	With default lower bounds of 1. Always column-major storage.
Pascal	*x* : **ARRAY** [*a* ... *a*+*I*–1, *b* ... *b*+*j*–1] **OF** *type*	Always row-major storage.

Compiling BASIC Modules for Row-Major Array Storage

When you compile a BASIC program with the BC compiler, you can select the /R compile option, which specifies that row-major order is to be used, rather than the default column-major order. BASIC is the only Microsoft language that permits you to specify how arrays should be stored. The /R option is available only when compiling from the command line. The only choice for array storage in QBX (or when compiling from within QBX using the Make EXE command) is column-major storage.

Array Data in Memory

The following code could be used in BASIC to fill a 4 x 5 array with integers from 0 through 19:

```
DIM Arr%(3,4) AS INTEGER
FOR I% = 0 to 3
   FOR J% = 0 to 4
          Arr%(I%, J%) = number%
          PRINT Arr%(I%, J%)
          Number%= Number%+1
   NEXT J%
NEXT I%
```

Because BASIC uses column-major array storage by default, the numbers would be stored in contiguous memory locations. However, the order of the data, as placed in memory, is not the same as the numeric sequence (0–19) of the data. Because the arrangement in memory is column major, it would be as shown in Figure 12.11.

0	5	10	15	1	6	11	16	2	7	12	17	3	8	13	18	4	9	14	19

Figure 12.11 Column-Major Storage

However, if compiled from the command line with BASIC's /R option, the array shown in the preceding section would be stored as shown in Figure 12.12:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Figure 12.12 Row-Major Storage

Within the QBX environment only the default can be used; the /R option is not available.

Passing Arrays Between Modules

Only BASIC allows you to choose the order in which arrays are stored. Therefore, when you pass an array from BASIC to a language that expects arrays to be stored in row-major order, the easiest thing to do is to use the /R option when compiling the BASIC module. The next easiest thing to do is to reverse the order of the dimensions in the array declaration of one of the languages. For example, if you wanted to pass the BASIC array described in the preceding section to C function contained in a Quick library loaded in the QBX environment, this would be the only option.

Given the address of the first element of the BASIC array shown in the preceding BASIC code, a C function could use the following code to print out its elements in the order in which they would be printed out in the BASIC code:

```
int arr[5][4] ;                          /* Note the number of elements = 5 */
int number = 0   , I = 0, j = 0 ;        /* in the major dimension, but the */
for (I =0; I = 4 ; I++)                  /* major-dimension upper bound = 4 */
   for (j =0; j = 3 ; j++)
         printf( "%d ", arr[I][j] ) ;
```

The data stored by BASIC is not actually reordered by the C code. Instead the dimensions in the C array are declared in reverse order from those in the BASIC declaration. To access an element in an array, compilers use a formula similar to the following:

starting address + ((*MajorUpperBound* * *MajorSubscript*) + *MinorSubscript*) * *scale*

MajorUpperBound is the number of elements in the major dimension (in the default BASIC case, the right, or column dimension). *MajorSubscript* is the actual major index value of the element you want to access (with the BASIC default, the right index value). *MinorSubscript* is the actual minor index value of the element you want to access (in the case of BASIC, the left index value). The *scale* is the size of each data element, for example an integer is 2 bytes, a long integer is 4 bytes, etc.

Example The following references all refer to the same place in memory for an array:

```
arr1[2][8]/* in C */

Arr1[3,9]        {  in Pascal, assuming lower bounds of 1  }

ARR1(8,2)     ' in BASIC, assuming default array storage & C lower bounds

C                In FORTRAN, assuming lower bounds of 1
                 ARR1(9,3)
```

Arrays with More than Two Dimensions

Describing arrays in terms of rows and columns is a convenient analogy for understanding two-dimensional arrays. Actual storage is simply by contiguous memory locations. However, the format of the declarations shown earlier in Table 12.6 can be extended to any number of dimensions that you may use. For example, the following C declaration:

```
int arr1[2][10][15][20] ;
```

Is equivalent to the following BASIC declaration:

```
DIM Arr1%(19, 14, 9, 1)
```

These are equivalent in the sense that the C array element represented by the following:

```
arr1[k][l][m][n]
```

Refers to the same memory location as the following BASIC array element:

```
Arr1(n, m, l, k)
```

Structures, Records, and User-Defined Types

The C **struct** type, the BASIC user-defined type, the FORTRAN record (defined with the **STRUCTURE** keyword), and the Pascal **record** type are equivalent. Therefore, these data types can be passed between C, FORTRAN, Pascal, and BASIC.

These types can be affected by the storage method. By default, C, FORTRAN, and Pascal use word alignment for types shorter than one word (type **char** and **unsigned char**). This storage method specifies that occasional bytes can be inserted as padding so that word and double-word data items start on an even boundary. (In addition, all nested structures and records start on a word boundary.)

If you are passing a structure or record across a mixed-language interface, your calling routine and called routine must agree on the storage method and parameter-passing convention. Otherwise, data will not be interpreted correctly.

Because Pascal, FORTRAN, and C use the same storage method for structures and records, you can exchange data between routines without taking any special precautions unless you modify the storage method. Make sure the storage methods agree before changing data between C, FORTRAN, and Pascal.

BASIC packs user-defined types, so your other-language routines must also pack structures (using the /Zp command-line option or the **pack** pragma, in C) to agree. Figure 12.13 contrasts packed and word-aligned storage.

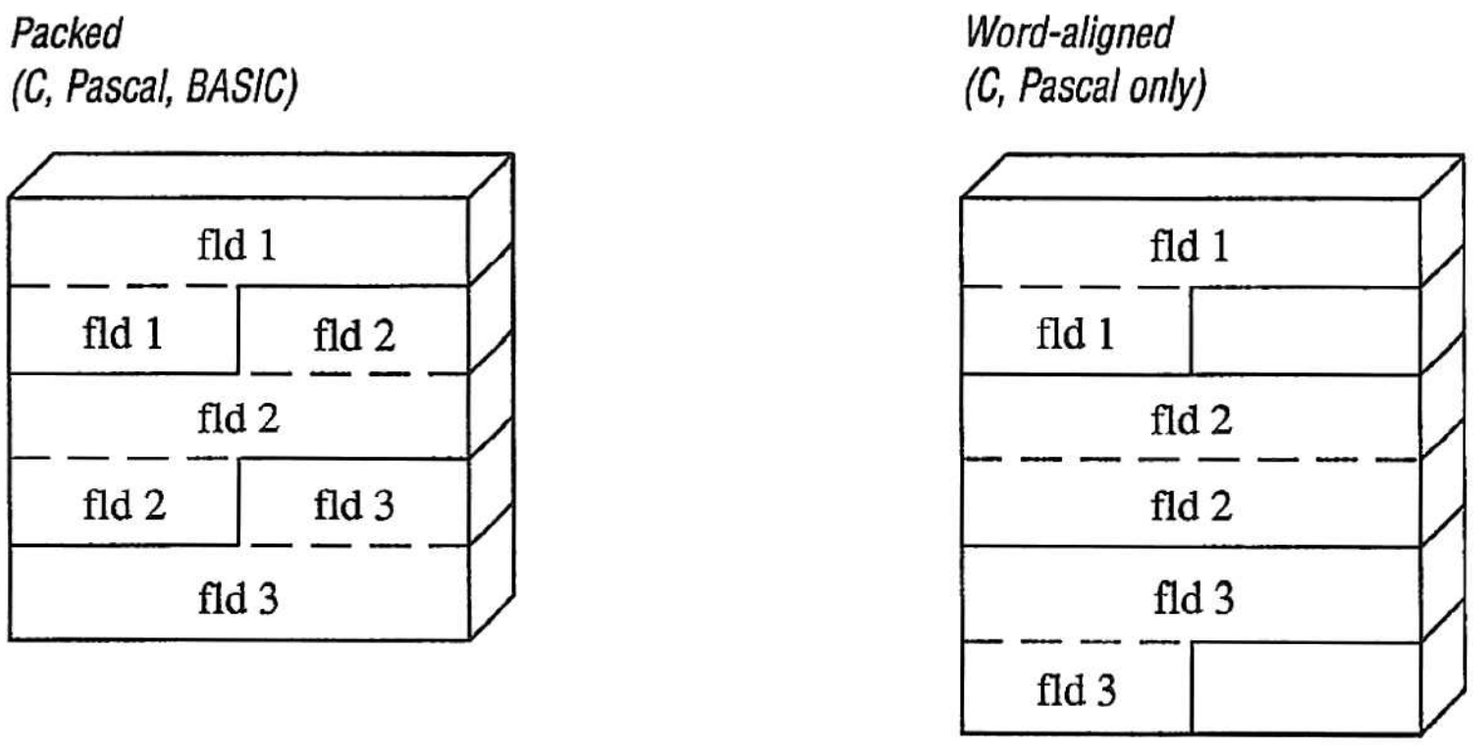


Figure 12.13 Structure and Record Storage

In C and Pascal, you can pass structures as parameters by value or by reference. In BASIC, structures (called user-defined types) can only be passed by reference. Both the calling program and the called program must agree on the parameter-passing convention. See the section "Parameter-Passing Requirements" earlier in this chapter for more information about the language you are using.

External Data

External data refers to data that is static and public; that is, the data is stored in a set place in memory as opposed to being allocated on the stack, and the data is visible to other modules. Although BASIC has static data, BASIC has no support for public data. Therefore there is no truly external data in BASIC. (See the section "Common Blocks" later in this chapter for more information about sharing external data with BASIC and FORTRAN programs.)

Pointers and Address Variables

Rather than passing data directly, you may want to pass the address of a piece of data. Passing the address amounts to passing the data by reference. In some cases, such as in BASIC arrays, there is no other way to pass a data item as a parameter.

BASIC and FORTRAN do not have formal address types. However, they do provide ways for storing and passing addresses.

BASIC programs can access a variable's segment address with the **VARSEG** function and its offset address with the **VARPTR** function. The values returned by these intrinsic functions should then be passed or stored as ordinary integer variables. If you pass them to another language, pass by value. Otherwise you will be attempting to pass the address of the address, rather than the address itself. See Chapters 13, "Mixed-Language Programming with Far Strings," and 11, "Advanced String Storage," for information on passing addresses of far strings.

To pass a near address, pass only the offset; if you need to pass a far address, you may have to pass the segment and the offset separately. Pass the segment address first, unless you have used **CDECL** in the BASIC **DECLARE** statement.

FORTRAN programs can determine near and far addresses with the **LOC** and **LOCFAR** functions. Store the result of the **LOC** function as **INTEGER*2** and the result of the **LOCFAR** function as **INTEGER*4**. As with BASIC, if you pass the result of **LOC** or **LOCFAR** to another language, be sure to pass by value.

C programs always pass array variables by address. All other types are passed by value unless you use the address-of (&) operator to obtain the address.

The Pascal **ADR** and **ADS** types are equivalent to C's near and far pointers, respectively. You can pass **ADR** and **ADS** variables as **ADRMEM** or **ADSMEM**.

Common Blocks

You can pass individual members of a BASIC or FORTRAN common block in an argument list, just as you can with any data. However, you can also give a different language module access to the entire common block at once.

C modules can refer to the items of a common block by first declaring a structure or record with fields that correspond to the common-block variables. Having defined a structure or user-defined type with the appropriate fields, the C module must then connect with the common block itself.

To pass the address of a common block, simply pass the address of the first variable in the block. (In other words, pass the first variable by reference.) The receiving C module should expect to receive a structure by reference.

Example In the following example, the C function `initcb` receives the address of the variable `N`, which it considers to be a pointer to a structure with three fields:

```
' BASIC SOURCE CODE
'
 COMMON /Cblock/ N%,X#,Y#
 DECLARE SUB INITCB CDECL (N%)
.
.
.
 CALL INITCB(N%)
.
.
.
/* C source code */
struct block_type {
 int N;
 double x;
 double y;
};
void initcb(block_hed)
struct block_type *block_hed;
{
 block_hed->n = 1;
 block_hed->x = 10.0;
 block_hed->y = 20.0;
}
```

Using a Varying Number of Parameters

Some C functions, most notably **printf**, can be called with a different number of arguments each time. To call such a function from another language, you need to suppress the type-checking that normally forces a call to be made with a fixed number of parameters. In BASIC, you can remove this type checking by omitting the parameter list from the **DECLARE** statement, as noted in the section "Alternative BASIC Interfaces" earlier in this chapter.

In FORTRAN or Pascal you can call routines with a variable number of parameters by using the **VARYING** attribute in your interface to the routine along with the **C** attribute. You must use the **C** attribute because a variable number of parameters is feasible only with the C calling convention.

Because the number of parameters is not fixed, the routine you call should have some mechanism for determining how many parameters to expect. Often this information is indicated by the first parameter. For example, the C function **printf** scans the format string passed as the first parameter. The number of fields in the format string determines how many additional parameters the function should expect. The following examples illustrate two ways of calling the C-library function **printf**.

Examples

In the first example a fixed number of arguments is declared, and the keyword **BYVAL** precedes each parameter in the **DECLARE** statement to insure that the true address of the string argument is passed. The example assumes you are using near strings in BASIC and compiling the C module in medium model.

```
DEFINT A-Z
DECLARE SUB printf CDECL (BYVAL Format, BYVAL Value)
String1$ = "Value passed to the formatted string is %d"+CHR$(0)
Number = 19
CALL printf(SADD(String1$), Number)
END
```

The following variation of the preceding example uses a special form of the **DECLARE** statement to declare the **printf** function. The parentheses that normally contain the formal parameters are omitted from the declaration. This informs BASIC of two facts:

- The procedure is not written in BASIC.
- The procedure can have a variable number of arguments. The **BYVAL** keyword is still used in the program, but this time it is passed with each argument in the **printf** call.

  In both examples, the C-language format character **%d** is used in the string passed to **printf**, which replaces it with the value of the second argument.

```
DEFINT A-Z
DECLARE SUB printf CDECL
String1$ = "Value passed to the formatted string is %d"+CHR$(0)
Number = 19
CALL printf(BYVAL SADD(String1$), BYVAL Number)
END
```

The preceding example describes the BASIC call to **printf**. When you link the object file created by such a program from the command line, **LINK** resolves the reference to the **printf** function when the executable file is created. However, if you want to call **printf** from within the QBX environment, you must make the function available within a Quick library. The simplest way to do this is to write a "dummy" C function that calls **printf**, and compile it (make sure you are compiling in medium model, as noted previously). Then, instead of linking it with a BASIC program on the link command line, incorporate the object file into a Quick library as shown in the following example:

```
LINK /Q DUMMY.OBJ,,,QBXQLB.LIB /NOE;
```

In this case a Quick library called `DUMMY.QLB` is created; it contains only the **printf** function. (Note that **LINK** will prompt you for the path to the correct C library if it cannot find it.) The /NOE option keeps **LINK** from misinterpreting certain symbols common to the two libraries as duplicate definitions. You can then make calls to printf if you load this library when you invoke BASIC, as follows:

```
QBX /L DUMMY
```

Using this method, you can create Quick libraries that include useful functions from the standard C library, as well as other-language routines you write yourself. See Chapter 19, "Creating and Using Quick Libraries," for more information.

B_OnExit Routine

You can use **B_OnExit** when your other-language routines take special actions that need to be undone before program termination or rerunning of the program. For example, within the BASIC environment, an executing program that calls other-language routines in a Quick library may not always run to normal termination. If such routines need to take special actions at termination (for example, de-installation of previously installed interrupt vectors), you can guarantee that your termination routines will always be called if you include an invocation of **B_OnExit** in the routine. The following example illustrates such a call (for simplicity, the example omits error-handling code). Note that such a function would be compiled in C in large model.

```
#include <malloc.h>
extern pascal far B_OnExit(); /* Declare the routine */
int *p_IntArray;
void InitProc()
{
 void TermProc();              /* Declare TermProc function */

 /* Allocate far space for 20-integer array: */
 p_IntArray = (int *)malloc(20*sizeof(int));
```

```
 /* Log termination routine (TermProc) with BASIC: */
 B_OnExit(TermProc);
}

                                /* The TermProc function is */
void TermProc()                 /* called before any restarting */
{                               /* or termination of program. */
 free(p_IntArray);              /* Release far space allocated */
}                               /* previously by InitProc. */
```

If the `InitProc` function were in a Quick library, the call to **B_OnExit** would insure proper release of the space reserved in the call to **malloc**, should the program end before normal termination could be performed. The routine could be called several times, since the program can be executed several times from the BASIC environment. However, the `TermProc` function itself would be called only once each time the program runs.

The following BASIC program is an example of a call to the `InitProc` function:

```
DECLARE SUB InitProc CDECL
X = SETMEM(-2048)               ' Make room for the malloc memory
                                ' allocation in C function.
CALL InitProc
END
```

If more than 32 routines are registered, **B_OnExit** returns NULL, indicating there is not enough space to register the current routine. Note that **B_OnExit** has the same return values as the Microsoft C run-time library routine **onexit.**

B_OnExit can be used with assembly language or any other-language routines you place in a Quick library. With programs compiled and linked completely from the command line, **B_OnExit** is optional.

Assembly Language-to-BASIC Interface

With MASM you can write assembly language modules that can be linked to modules developed with Microsoft BASIC, Pascal, FORTRAN, and C. This section outlines the recommended programming guidelines for writing assembly language routines compatible with Microsoft BASIC.

Writing assembly language routines for Microsoft high-level languages is easiest when you use the simplified segment directives provided with the MASM version 5.0, and later. This manual assumes that you have version 5.0 or later.

Writing the Assembly Language Procedure

You can call assembly language procedures using essentially the same conventions as for compiler-generated code. This section describes how you use those conventions to call assembly language procedures. Procedures that observe these conventions can be called recursively and can be debugged with the Microsoft CodeView debugger. The standard assembly language interface method, described in the following sections, consists of the following steps:

1. Set up the procedure.
2. Enter the procedure.
3. Allocate local data (optional).
4. Preserve register values.
5. Access parameters.
6. Return a value (optional).
7. Exit the procedure.

Note

The MASM, version 5.0 and later, provide an include file, MIXED.INC, that automatically performs many of the stack-maintenance chores described in the next few sections. Version 5.1 and later include several keywords that simplify creation of a mixed-language interface. Part 1 of the *Microsoft QuickAssembler Programmer's Guide* describes the mechanics of mixed-language programming using the mixed-language keywords.

Setting Up the Procedure

LINK cannot combine the assembly language procedure with the calling program unless compatible segments are used and unless the procedure itself is declared properly. The following four points may be helpful:

- Use the **.MODEL** directive at the beginning of the source file. This directive automatically causes the appropriate kind of returns to be generated (**NEAR** for small or compact model, **FAR** otherwise). Modules called from BASIC should be **.MODEL MEDIUM**. If you have a version of the MASM previous to 5.0, declare the procedure **FAR**.
- Use the simplified segment directives **.CODE** to declare the code segment and **.DATA** to declare the data segment. (Having a code segment is sufficient if you do not have data declarations.) If you are using a version of the assembler earlier than 5.0, declare the segments using the **SEGMENT**, **GROUP**, and **ASSUME** directives (described in the Microsoft Macro Assembler manual).
- Use the **PUBLIC** directive to declare the procedure label public. This declaration makes the procedure visible to other modules. Also, any data you want to make public to other modules must be declared as **PUBLIC**.

- Use the **EXTRN** directive to declare any global data or procedures accessed by the routine as external. The safest way to use code **EXTRN** declarations is to place the directive outside of any segment definition. However, place near data inside the data segment.

Note

If you want to be able to call the assembly language procedure from both BASIC and C, you can create the common interface using the **CDECL** keyword in the BASIC declaration, then following the C naming and calling conventions (as explained in the section "Using **CDECL** in calls from BASIC" earlier in this chapter).

Entering the Procedure

If your procedure accepts arguments or has local (automatic) variables on the stack, you need to use the following two instructions to set up the stack frame and begin the procedure:

```
push bp
mov bp,sp
```

This sequence establishes **BP** as the framepointer. The "framepointer" is used to access parameters and local data, which are located on the stack. The value of the base register **BP** should remain constant throughout the procedure (unless your program changes it), so that each parameter can be addressed as a fixed displacement off of **BP**. **SP** cannot be used for this purpose because it is not an index or base register. Also, the value of **SP** may change as more data is pushed onto the stack.

The instruction `push bp` preserves the value of **BP**. This value will be needed by the calling procedure as soon as the current procedure terminates. The instruction `mov bp,sp` captures the value that the stack pointer had at the time of entry to the procedure. This establishes that the parameter can be addressed.

Allocating Local Data (Optional)

An assembly procedure can use the same technique for allocating temporary storage for local data that is used by high-level languages. To set up local data space, simply decrease the contents of **SP** immediately after setting up the stack frame. To ensure correct execution, you should always increase or decrease **SP** by an even amount. Decreasing **SP** reserves space on the stack for the local data. The space must be restored at the end of the procedure as shown by the following:

```
push bp
mov bp,sp
sub sp,space
```

In the preceding code fragment, space is the total size in bytes of the local data. Local variables are then accessed as fixed, negative displacements off of **BP**.

Example The following example uses two local variables, each of which is 2 bytes. **SP** is decreased by 4, since there are 4 bytes total of local data. Later, each of the variables is initialized to 0.

```
push bp          ; Save old stack frame
mov bp,sp        ; Set up new stack frame
sub sp,4         ; Allocate 4 bytes local storage

mov WORD PTR [bp-2],0
mov WORD PTR [bp-4],0
```

Local variables are also called dynamic, stack, or automatic variables.

Preserving Register Values

A procedure called from any of the Microsoft high-level languages should preserve the direction flag and the values of **SI**, **DI**, **SS**, and **DS** (in addition to **BP**, which is already saved). Any register values that your procedure alters should be pushed onto the stack after you set up the stack frame, but before the main body of the procedure. If the procedure does not change the value of any of these registers, then the registers do not need to be pushed.

Warning Routines that your assembly language procedure calls must not alter the **SI**, **DI**, **SS**, **DS**, or **BP** registers. If they do, and you have not preserved the registers, they can corrupt the calling program's register variables, segment registers, and stack frame, causing program failure.

If your procedure modifies the direction flag using the **STD** or **CLD** instructions, you must preserve the flags register.

The following example shows an entry sequence that sets up a stack frame, allocates 4 bytes of local data space on the stack, then preserves the **SI**, **DI**, and flags registers.

```
push    bp          ; Save caller's stack frame.
mov     bp,sp       ; Establish new stack frame.
sub     sp,4        ; Allocate local data space.
push    si          ; Save SI and DI registers.
push    di
pushf               ; Save the flags register.
.
.
.
```

In the preceding example, you must exit the procedure with the following code:

```
popf                    ; Restore the flags register.
pop     di              ; Restore the old value in the DI register.
pop     si              ; Restore the old value in the SI register.
mov     sp,bp           ; Restore the stack pointer.
pop     bp              ; Restore the frame pointer.
ret                     ; Return to the calling routine.
```

If you do not issue the preceding instructions in the order shown, you will place incorrect data in registers. The rules for restoring the calling program's registers, stack pointer, and frame pointer are:

- Pop all registers that you preserve in the reverse order from which they were pushed onto the stack. So, in the example above, **SI** and **DI** are pushed, and **DI** and **SI** are popped.
- Restore the stack pointer by transferring the value of **BP** into **SP** before restoring the value of the frame pointer.
- Always restore the frame pointer last.

Accessing Parameters

Once you have established the procedure's framepointer, allocated local data space (if desired), and pushed any registers that need to be preserved, you can write the main body of the procedure. To write instructions that can access parameters, consider the general picture of the stack frame after a procedure call, as illustrated in Figure 12.14.

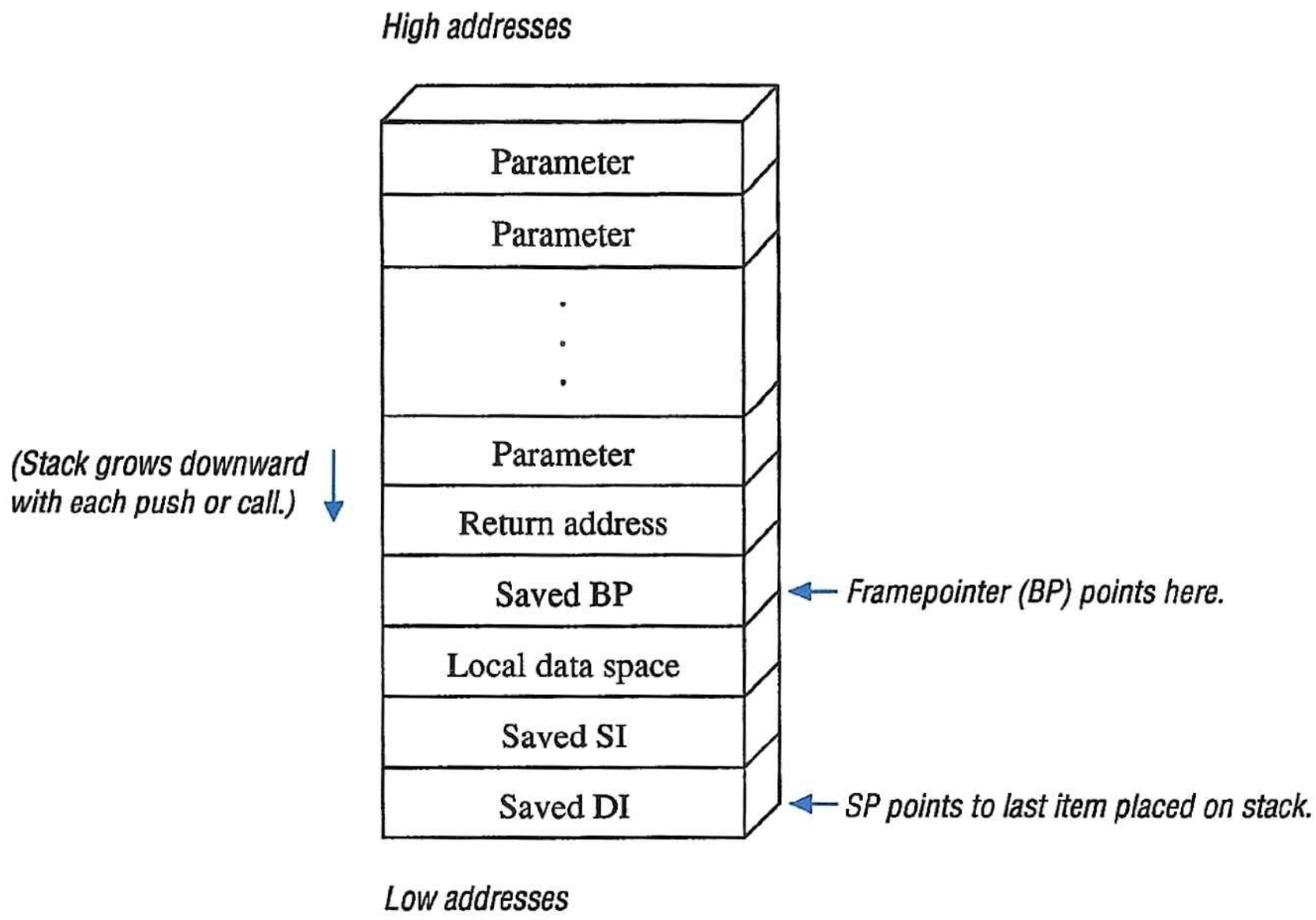


Figure 12.14 The Stack Frame

The stack frame for the procedure is established by the following sequence of events:

1. The calling program pushes each of the parameters on the stack, after which **SP** points to the last parameter pushed.
2. The calling program issues a **CALL** instruction, which causes the return address (the place in the calling program to which control ultimately returns) to be placed on the stack. Because BASIC always uses a **FAR** call, this address is 4 bytes long. **SP** now points to this address. With a language such as C, the address may be 2 bytes, if the call is a near call.
3. The first instruction of the called procedure saves the old value of **BP**, with the instruction `push bp`. Now **SP** points to the saved copy of **BP**.
4. **BP** is used to capture the current value of **SP**, with the instruction `mov bp,sp`. **BP** therefore now points to the old value of **BP** (saved on the stack).
5. Whereas **BP** remains constant throughout the procedure, **SP** is often decreased to provide room on the stack for local data or saved registers.

In general, the displacement (off of **BP**) for a parameter X is equal to 2 plus the size of return address plus the total size of parameters between X and **BP**.

For example, consider a procedure that has received one parameter, a 2-byte address. Since the size of the return address is always 4 bytes in BASIC, the displacement of the parameter would be calculated as follows:

argument's displacement = 2 + *size of return address*

```
= 2 + 4
= 6
```

In other words, the argument's displacement equals 2 plus 4, or 6. The argument can thus be loaded into **BX** with the following instruction:

```
mov bx,[bp+6]
```

Once you determine the displacement of each parameter, you may want to use the **EQU** directive or structures to refer to the parameter with a single identifier name in your assembly source code. For example, the preceding parameter at `BP+6` can be conveniently accessed if you put the following statement at the beginning of the assembly source file:

```
Arg1        EQU     [bp+6]
```

You could then refer to this parameter as `Arg1` in any instruction. Use of this feature is optional.

Note

For far (segment plus offset) addresses, Microsoft high-level languages push segment addresses before pushing offset address. Furthermore, when pushing arguments larger than 2 bytes, high-order words are always pushed before low-order words and parameters longer than 2 bytes are stored on the stack in most-significant, least-significant order. This standard for pushing segment addresses before pushing offset addresses facilitates the use of the **LES** (load extra segment) and **LDS** (load data segment) instructions.

Returning a Value (Optional)

BASIC has a straightforward convention for receiving return values when the data type to be returned is simple (that is, not a floating-point value, an array, or structured type) and is no more than 4 bytes. This includes all pointers and all parameters passed by reference, as shown in the following list:

Data type	Returned in register
2-byte integer(%)	**AX**
4-byte integer(&)	High-order portion in **DX**; low-order portion in **AX**
All other types	Near offset in **AX**

An assembly language procedure called by BASIC must use a special convention to return floating-point values, user-defined types and arrays, and values larger than 4 bytes, as explained in the following section:

Numeric Return Values Other Than 2- and 4-Byte Integers

In order to create an interface for numeric return values that are neither 2-byte integers (%) nor 4-byte integers (&), BASIC modules take the following actions before they call your procedure (assuming that the BASIC declarations do not specify the **CDECL** keyword):

1. When the call to your procedure is made, an extra parameter is passed; this parameter contains the offset address of the actual return value. This parameter is placed immediately above the return address. (In other words, this parameter is the last one pushed.)
2. The segment address of the return value is contained in **SS** and **DS**.

The extra parameter (which contains the offset address of the return value) is always located at **BP+6**. Furthermore, its presence automatically increases the displacement of all other parameters by two, as shown in Figure 12.15.

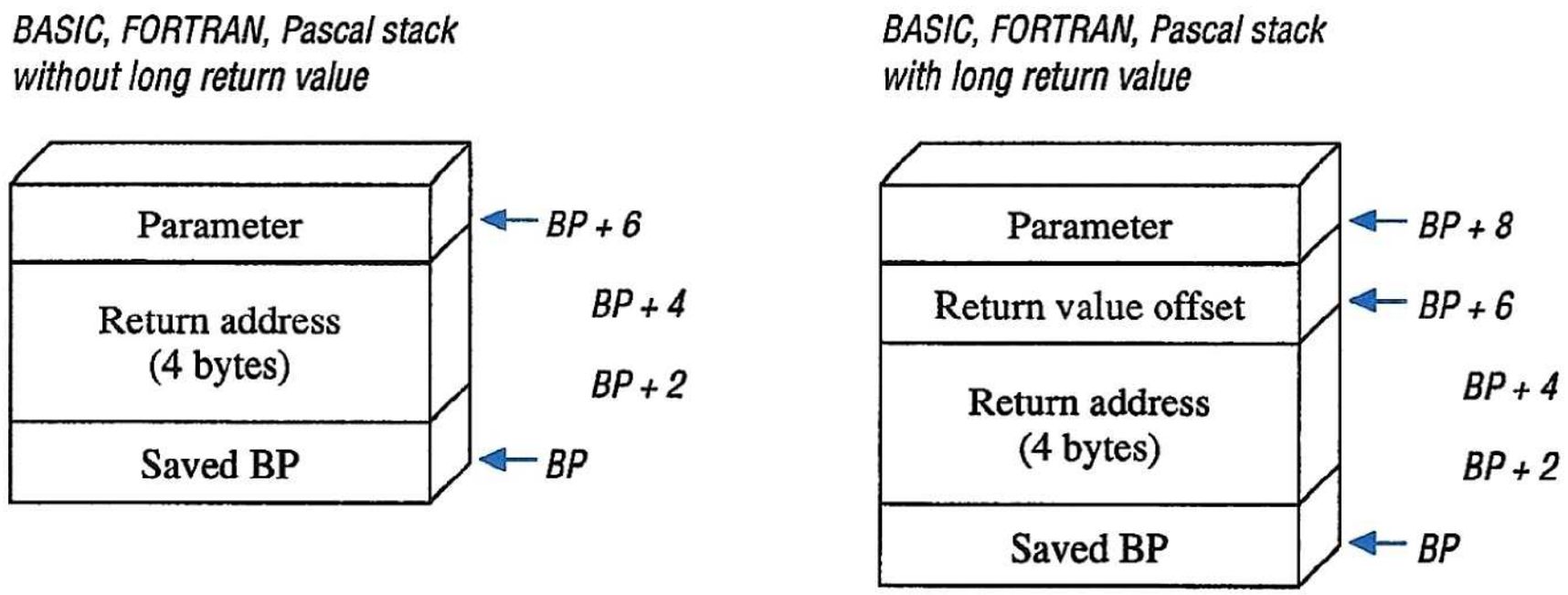


Figure 12.15 BASIC Return Values

Your assembly language procedure can successfully return numeric values other than 2- and 4-byte integers if you follow these steps:

1. Put the data for the return value at the location pointed to by the return-value offset.
2. Copy the return-value offset (located at **BP+6**) to **AX**. This is necessary because the calling module expects **AX** to point to the return value.
3. Exit the procedure as described in the next section.

Exiting the Procedure

Several steps may be involved in terminating the procedure:

1. If any of the registers **SS**, **DS**, **SI**, or **DI** have been saved, these must be popped off the stack in the reverse order from that in which they were saved. If they are popped in any other order, program behavior is unpredictable.
2. If local data space was allocated at the beginning of the procedure, **SP** must be restored with the instruction `mov sp,bp`.
3. Restore **BP** with the instruction `pop bp`. This step is always necessary.
4. Since the BASIC calling convention is being used, you must use the **RET** *n* form of the instruction to adjust the stack if any parameters were pushed by the caller.

Examples The following example shows the simplest possible exit sequence. No registers were saved, no local data space was allocated, and no parameters were passed to the routine.

```
pop bp
ret
```

The following example shows an exit sequence for a procedure that has previously saved **SI** and **DI**, allocated 4 bytes of local data space, used the BASIC calling convention, and received 6 bytes of parameters. The procedure must therefore use `ret 6` to restore the 6 bytes of parameters on the stack.

```
push    bp
mov     bp,sp
sub     sp,4
push    si
push    di
 .
 .
 .
pop     di              ; pop saved registers
pop     si
mov     sp,bp           ; Free local data space.
pop     bp              ; Restore old stack frame.
ret     6               ; Exit, and remove 6 bytes of args.
```

Note If the preceding routine had been declared with **CDECL**, only the **RET** (without a specification of *n*) would be used because with the C calling convention, the caller cleans up the stack.

Calls from BASIC

A BASIC program can call an assembly language procedure in another source file with the **CALL** or **CALLS** statement. Proper use of the **DECLARE** statement is also important. In addition to the steps outlined in the preceding sections, the following guidelines may be helpful:

- Declare procedures called from BASIC as **FAR**.
- Observe the BASIC calling conventions:
  - Parameters are placed on the stack in the same order in which they appear in the BASIC source code. The first parameter is highest in memory (because it is also the first parameter to be placed on the stack, and the stack grows downward).
  - By default, BASIC parameters are passed by reference as 2-byte addresses. (See Chapter 13, "Mixed-Language Programming with Far Strings," for information on passing strings stored in far memory.)
  - Upon exit, the procedure must reset **SP** to the value it had before the parameters were placed on the stack. This is accomplished with the instruction **RET** *n*, where *n* is the total size in bytes of all the parameters.
- Observe the BASIC naming convention.

BASIC outputs symbolic names in uppercase characters, which is also the default behavior of the assembler. BASIC recognizes up to 40 characters of a name, whereas the assembler recognizes only the first 31 (this should rarely create a problem).

Note Microsoft BASIC provides a Quick library (called QBX.QLB and an include file called QBX.BI). These files contain several routines that facilitate calling assembly language routines from within the BASIC environment. See Chapter 11, "Advanced String Storage," for information on using this library.

Examples In the following example, BASIC calls an assembly language procedure that calculates $A * 2^B$, where A and B are the first and second parameters, respectively. The calculation is performed by shifting the bits in A to the left, B times.

```
' BASIC program
'
DEFINT A-Z
'
DECLARE FUNCTION Power2(A%,B%)
'
PRINT "3 times 2 to the power of 5 is ";
PRINT Power2(3,5)
END
```

To understand how to write the assembly language procedure, recall how the parameters are placed on the stack, (shown in Figure 12.15):

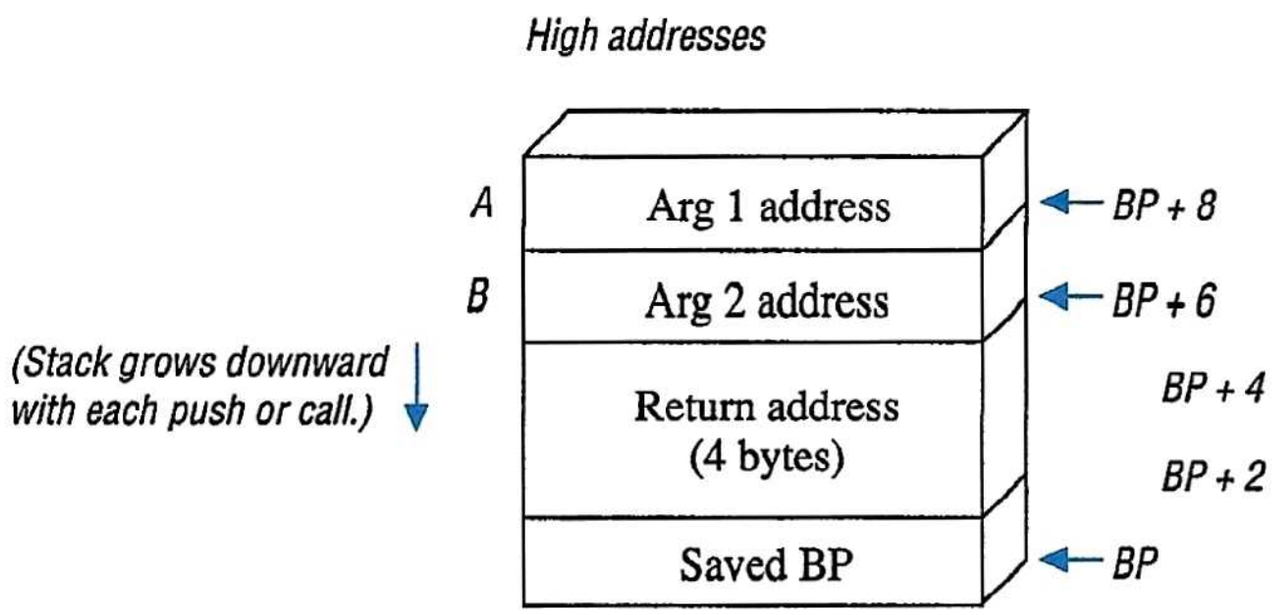


Figure 12.15 BASIC Stack Frame

The return address is 4 bytes because procedures that are called from BASIC must be **FAR**. Arg 1 is higher in memory than Arg 2 because BASIC pushes arguments in the same order in which they appear. Also, each argument is passed as a 2-byte offset address, the BASIC default.

The assembly language procedure can be written as follows:

```
.MODEL MEDIUM
.CODE
 PUBLIC Power2
Power2 PROC
 push     bp              ; Entry sequence - saved old BP
 mov      bp,sp           ; Set stack framepointer
                          ;
 mov      bx,[bp+8]       ; Set BX equal to address of Arg1
 mov      ax,[bx]         ; Load value of Arg1 into AX
 mov      bx,[bp+6]       ; Set BX equal to address of Arg2
 mov      cx,[bx]         ; Load value Arg2 into CX
 shl      ax,cl           ; AX = AX * (2 to power of CX)
                          ; Leave return value in AX
 pop      bp              ; Exit sequence - restore old BP
 ret      4               ; Return, and restore 4 bytes
Power2 ENDP
 END
```

Note that each parameter must be loaded in a two-step process because the address of each is passed rather than the value. Also, note that the stack is restored with the instruction `ret 4` since the total size of the parameters is 4 bytes. (The preceding example is simplified to illustrate the interlanguage interface. Code for handling possible errors, such as overflow, is not included, but is a significant consideration with such procedures.)

Using CDECL in Calls from BASIC

You can use the **CDECL** keyword in the **DECLARE** statement in your BASIC module to call an assembly language routine. If you do so, the C calling conventions, rather than those of BASIC, determine the order of the arguments as received in the assembly language routine, and also the manner in which returns are handled. The primary advantage of using **CDECL** is that then you can call the assembly language routine with a variable number of arguments. The technique is analogous to calling a C function from BASIC using **CDECL**. In using **CDECL**, observe the C calling convention:

- Parameters are placed on the stack in the reverse order to that in which they appear in the BASIC source code. This means the first parameter is lowest in memory (because the stack grows downward, and it is the last parameter to be placed on the stack).
- Return with a simple **RET** instruction. Do not restore the stack with **RET** *n*, since using **CDECL** causes the calling routine to restore the stack itself as soon as the calling routine resumes control. When the return value is not a 2-byte or 4-byte numeric value, a procedure called by BASIC with **CDECL** in effect must allocate space for the return value, and then place its address in **AX**. A simple way to create space for the return value is to declare it in a data segment.
- If **CDECL** is used in the BASIC **DECLARE** statement, you must name the assembler procedure with a leading underscore, unless you use the **ALIAS** feature.

The Microsoft Segment Model

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, versions of MASM prior to 5.0 do not support these directives. With older versions of the assembler, you should use the **SEGMENT**, **GROUP**, **ASSUME**, and **ENDS** directives equivalent to the simplified segment directives.

Table 12.7 shows the default segment names created by each directive for medium model, the only model applicable to BASIC. Use of these segments ensures compatibility with Microsoft languages and helps you to access public symbols. This table is followed by a list of three steps, illustrating how to make the actual declarations.

Table 12.7 Default Segments and Types for Medium Memory Model

Directive	Name	Align	Combine	Class	Group	Description of segment
CODE	*name*_TEXT	**WORD**	**PUBLIC**	**'CODE'**	—	The segment containing all the code for the module.
DATA	_DATA	**WORD**	**PUBLIC**	**'DATA'**	**DGROUP**	Initialized data.
CONST	**CONST**	**WORD**	**PUBLIC**	**'CONST'**	**DGROUP**	Uninitialized data. Microsoft compilers store uninitialized data separately because it can be more efficiently stored than initialized data.
DATA?	_BSS	**WORD**	**PUBLIC**	**'BSS'**	**DGROUP**	Constant data. Microsoft compilers use this segment for such items as string and floating-point constants.
STACK	**STACK**	**PARA**	**STACK**	**'STACK'**	**DGROUP**	Stack. Normally, this segment is declared in the main module for you and should not be redeclared.

The following steps describe how to use Table 12.7 to create directives:

1. Use Table 12.7 to determine the segment name, align type, combine type, and class for your code and data segments. Use all of these attributes when you define a segment. For example, the code segment is declared as follows:

   ```
   name_TEXT      SEGMENT          WORD    PUBLIC 'CODE'
   ```

   The `name_TEXT` and all the attributes are taken from Table 12.7. You substitute your module name for name. If the combine type is private, simply do not use any combine type.

2. If you have segments in **DGROUP**, put them into **DGROUP** with the **GROUP** directive, as in:

   ```
   GROUP          DGROUP          _DATA   _BSS
   ```

3. Use **ASSUME** and **ENDS** as you would normally. Upon entry, **DS** and **SS** both point to **DGROUP**; therefore, a procedure that makes use of **DGROUP** should include the following **ASSUME** directive:

   ```
   ASSUME CS:name_TEXT, DS:DGROUP,     SS:DGROUP
   ```

Note If your assembly language procedures use real numbers, they must use IEEE-format numbers to be compatible with QBX. This is the default for MASM, version 5.0 and later. With earlier versions, you must specify the **IR** command-line option or the **8087** directive.

Chapter 13 Mixed-Language Programming with Far Strings

This chapter provides new techniques for passing strings between BASIC and another language. These techniques supplement the general method of programming outlined in Chapter 12, "Mixed-Language Programming." To get the most out of the present chapter, you should understand the material presented in that guide—most importantly, be familiar with the naming, calling, and passing conventions of BASIC and the other languages you are using.

This chapter contains the following information:

- A description of BASIC's string-processing routines.
- A general string-passing model.
- Specific examples of passing strings between Microsoft BASIC, Macro Assembler (MASM), C, Pascal, and FORTRAN.

Considerations When Using Strings

When passing strings between BASIC and another language, several complications arise. First of all, BASIC's sending and receiving parameters can only be variable-length strings; other languages use fixed-length strings. Furthermore, whenever a BASIC **SUB** procedure is receiving a string, it expects to be passed the address of a variable-length string descriptor, a data structure unique to BASIC.

If the BASIC program is using far strings, a further complication arises because the structure of the far string descriptor is proprietary. It is not possible for the external routine to get information directly from the descriptor, or to create a far string descriptor for existing fixed-string data, and have BASIC recognize it as one of its own far strings.

To make it easier for you to pass strings in these situations, routines in the BASIC run-time and stand-alone libraries are provided with Microsoft BASIC. These routines are described in the next section. After that, a general method for passing strings is outlined. This same method works for near and far strings. Specific examples of using this method with MASM, C, Pascal, and FORTRAN are then given.

String-Processing Routines

The routines described in the following sections are designed to transfer string data between languages, deallocate string space, and to determine the length and location of variable-length strings. The routines are summarized in the following table.

Table 13.1 String-Processing Routines

Routine name	Purpose
StringAssign	Transfers string data from one language's memory space to another.
StringRelease	Deallocates string data space created by an external language using **StringAssign.**
StringAddress	Returns a far pointer to BASIC string data.
StringLength	Returns the length of a BASIC string.

The routines can be used by any language including BASIC and behave like any other external call. (See the **DECLARE** statement in the *BASIC Language Reference* for more information.) They are declared as external procedures and loaded from the appropriate BASIC run-time or stand-alone library during linking. For running and debugging within QBX, they need to be in a Quick library. The section "Passing Strings in QBX" later in this chapter describes how to do this.

Transferring String Data

The **StringAssign** routine lets you transfer string data from one language memory space to another. Typically it is used to transfer a BASIC variable-length string to a second language's fixed-length string and vice versa. The syntax is:

CALL StringAssign(*sourceaddress&, sourcelength%, destaddress&, destlength%*)

The *sourceaddress&* argument is a far pointer to the string descriptor if the source is a variable-length string, or it is a far pointer to the start of string data if the source is a fixed-length string. The *sourcelength%* argument is 0 for variable-length strings, otherwise it contains the length of the fixed-length string source in bytes. The *destaddress&* argument is a far pointer to the string descriptor if the destination is a variable-length string, or it is a far pointer to the start of string data if the destination is a fixed-length string. The *destlength%* argument is 0 for variable-length strings, otherwise it contains the length of the fixed-length string destination.

Arguments are passed to **StringAssign** by value using the **BYVAL** keyword. The following is an example of the declaration:

```
DECLARE SUB StringAssign(BYVAL Src&, BYVAL SrcLen%, BYVAL Dest&,_
                         BYVAL DestLen%)
```

When far pointers are not available, they can be generated by passing the segment and the offset separately. For example, assume that the segment, offset, and length of an external fixed-length string are contained in the variables `FixedSeg%`, `FixedOff%`, and `FixedLength%`. The string can be assigned to variable-length string `A$` as follows:

```
DECLARE SUB StringAssign(BYVAL SrcSeg%,BYVAL SrcOff%,BYVAL SrcLen%,_
                         BYVAL DestSeg%,BYVAL DestOff%,BYVAL DestLen%)
CALL StringAssign(FixedSeg%,FixedOff%,FixedLength%,VARSEG(A$),_
                  VARPTR(A$),0)
```

To assign the variable-length string `A$` back to the fixed-length string, BASIC does the reverse:

```
CALL StringAssign(VARSEG(A$),VARPTR(A$),0,FixedSeg%,FixedOff%,_
                  FixedLength%)
```

MASM, C, Pascal, and FORTRAN deal only with fixed-length strings. When programming in these languages you can use **StringAssign** to create a new BASIC variable-length string and transfer fixed-string data to it. To transfer a MASM string containing the word "hello" to a BASIC string, for example, you use this data structure:

```
fixedstring  db  "Hello"              ; source of data
descriptor   dd  0                    ; descriptor for destination
```

The second data element, descriptor, is a 4-byte string descriptor initialized to zero. BASIC interprets this to mean that it should create a new variable-length string and associate it with the address descriptor. Assigning the fixed-length string to it is accomplished by the following:

```
push  ds                              ;segment of near fixed string
lea   ax, fixedstring                 ;offset of near fixed string
push  ax
mov   ax, 5                           ;fixed string length
push  ax
push  ds                              ;segment of descriptor
lea   ax, descriptor                  ;offset of descriptor
push  ax
xor   ax, ax                          ;0 means that destination
push  ax                              ;is a variable-length string
extrn stringassign:  proc far         ;transfer the data
call  stringassign
```

When the call to **StringAssign** is made, BASIC will fill in the double-word descriptor with the correct string descriptor.

Note

When creating a new variable-length string you must allocate 4-bytes of static data for a string descriptor as shown in the preceding procedure. Allocating the data on the stack will not work.

A new variable-length string that is created with **StringAssign** can be used as an argument to a procedure. The address of the string descriptor is pushed on the stack and thus becomes a procedure parameter that can be processed like any other variable-length string. Examples of creating strings in all languages with the **StringAssign** routine can be found in the code examples which follow this section.

Deallocating String Data

The **StringRelease** routine deallocates variable-length strings created in a non-BASIC language by **StringAssign**. This frees the space in BASIC's string data area. You can use this routine to deallocate strings after your string processing tasks are completed. **StringRelease** has the following syntax:

CALL StringRelease(*string-descriptor%*)

The *string-descriptor%* argument is a near pointer to the variable-length string descriptor. The pointer is passed by value.

To perform the release with MASM, assuming a descriptor for the variable-length string `NoLongerNeeded` exists at offset `descriptor1`, write the following code:

```
lea    ax, descriptor1
push   ax
extrn stringrelease: proc far
call   stringrelease
```

Important

StringRelease is only for variable-length strings created by a language other than BASIC. Never use it on strings that are created by BASIC. Doing so will cause unpredictable results.

Computing String Data Addresses

StringAddress is a routine that returns a far pointer to variable-length string data. It is the equivalent of **SSEGADD**. Assume that a descriptor for a BASIC variable-length string `MissingString$` exists at offset `descriptor1`. MASM can find the far address of the string data with the following fragment:

```
lea    ax, descriptor1
push   ax
extrn stringaddress: proc far
call   stringaddress
```

The far pointer is returned in `DX:AX`. `DX` holds the segment and `AX` holds the offset.

Computing String Data Length

StringLength is a routine function that returns the length of variable-length string data. It is the equivalent of **LEN**. Assume that a descriptor for a BASIC variable-length string `LongString$` exists at the label `descriptor1`. MASM can find the length of string data with the following fragment:

```
lea    ax, descriptor1
push   ax
extrn stringlength: proc far
call   stringlength
```

The length of `LongString$` is returned in AX.

Passing Variable-Length Strings

Any time you need to pass variable-length strings from one language to another, process them, and then pass the string data back for further processing, the following general method may prove useful. This method uses the **StringAssign** routine which works for any calling language and also works whether you are passing near or far strings.

1. The first module makes the call, providing string pointers (and length if calling an external procedure).
2. The second module assigns the data to its own string type using the **StringAssign** routine.
3. The second module processes the data.
4. The second module assigns the output string data to the first module's string type using the **StringAssign** routine.
5. The second module returns from the call, providing string pointers (and length if returning to an external procedure).
6. The first module continues processing.

BASIC Calling MASM

This example shows how to pass variable-length strings between BASIC and MASM using the general method explained previously. The first module, MXSHKB.BAS, creates the strings `A$` and `B$`, then passes their string-descriptor far pointers and data lengths to a MASM procedure named `AddString`. This procedure is contained in the file MXADSTA.ASM. The `AddString` procedure transfers the data to its own work area and then concatenates the two strings. Finally, `AddString` transfers the output to a BASIC variable-length string. Upon return, the BASIC module prints the output string.

Important **StringAssign, StringRelease, StringAddress,** and **StringLength** may change the contents of the **AX, BX, CX, DX,** and **ES** registers and may change any flags other than the direction flag. These registers and flag should be saved before calling any of these string routines if their values are important to your MASM program.

This MASM code uses the **.MODEL** directive which establishes compatible naming, calling, and passing conventions for BASIC, and it also uses simplified segment directives. This eliminates the need for separate **GROUP** and **ASSUME** directives. See Section 8.2, "Declaring Symbols External," of the *Microsoft Macro Assembler 5.1 Programmer's Guide* manual for a comparison of this method with one using full segment definitions. The version 5.1 **PROC** directive is employed. It includes new arguments that specify automatically saved registers, define arguments to the procedure, and set up text macros to use for the arguments. The **PROC** directive automatically generates the proper type of return based on the chosen memory model and cleans up the stack.

```
'*******************************MXSHKB.BAS*******************************
DEFINT A-Z

'Define a non-BASIC procedure.
DECLARE FUNCTION AddString$(SEG S1$,BYVAL S1Length,SEG S2$,BYVAL S2Length)

'Create the data.
A$ = "To be or not to be;"
B$ = " that is the question."

'Use non-BASIC function to add two BASIC far strings.
C$ = (A$, LEN(A$), B$, LEN(B$))

'Print the result on the screen.
PRINT C$
```

```
;***************************** MXADSTA.ASM ****************************
;  This procedure accepts two far strings, concatenates them, and
;  returns the result in the form of a far string.
;
        .model medium,basic     ;define memory model to match BASIC
        .stack
        .data?
       maxst = 50          ;maximum bytes reserved for strings
inbuffer1 db  maxst dup(0)      ;room for first fixed-length string
inbuffer2 db  maxst dup(0)      ;and second one
outbuffer db  2*maxst dup(0)    ;work area for string processing
        .data
sh          dd  0               ;output string descriptor
        .code
addstring   procuses si di ds, s1:far ptr, s1len, s2:far ptr, s2len

;First get BASIC to convert BASIC strings into standard form:
        les     ax,s1                       ;Push far pointer
        push    es                          ;to input string
        push    ax                          ;descriptor.
        xor     ax,ax                       ;Push a zero to indicate
        push    ax                          ;it is variable length.
        push    ds                          ;Push far pointer
        lea     ax, inbuffer1               ;to destination
        push    ax                          ;string.
        mov     ax,maxst                    ;Push length of destination
        push    ax                          ;fixed-length string.
        extrn   stringassign:proc far
        call    stringassign                ;Call BASIC to assign
                                            ;variable-length string
                                            ;to fixed length string.
        les     ax,s2                       ;Push far pointer to
        push    es                          ;second input string
        push    ax                          ;descriptor.
        xor     ax,ax                       ;Push a zero to indicate
        push    ax                          ;it is variable length.
        push    ds                          ;Push far pointer
        lea     ax,inbuffer2                ;to second
        push    ax                          ;destination string.
        mov     ax,maxst                    ;Push length of destination
        push    ax                          ;fixed-length string.
```

```
extrn   stringassign:proc far
call    stringassign            ; Call BASIC to assign
                                ; variable-length string
                                ; to fixed length string.
                                ; Concatenate strings:
lea     si,inbuffer1            ; Copy first string to buffer.
lea     di,outbuffer
mov     ax,ds
mov     es,ax
mov     cx,s1len
rep     movsb
lea     si,inbuffer2            ; Concatenate second string to
mov     cx,s2len                ; end of first.
rep     movsb

;Get BASIC to convert result back into a BASIC string:
        push    ds                      ;Push far pointer to
        lea     ax,outbuffer            ;fixed-length result
        push    ax                      ;string.
        mov     ax,s1len                ;Compute total length
        mov     bx,s2len                ;of fixed-length
        add     ax,bx                   ;result string.
        push    ax                      ;Push length.
        push    ds                      ;Push far pointer
        lea     ax,sh                   ;to sh (BASIC will use
        push    ax                      ;this in StringAssign).
        xor     ax,ax                   ;Push a zero for length
        push    ax                      ;indicating variable-length.
        call    stringassign            ;Call BASIC to assign the
                                        ;result to sh.
        lea     ax,sh                   ;Return output string pointer
                                        ;in ax and go back to BASIC.
        ret

addstring endp
        end
```

When returning to BASIC with only one string output, it is convenient for the MASM procedure to be a function, as in the preceding example. To pass back more than one string, do the following:

1. Have BASIC declare the procedure as a BASIC **SUB** procedure with output parameters included in the calling list:

```
DECLARE SUB AddString (SEG S1$, BYVAL S1Length, SEG S2$,_
                       BYVAL S2Length, OutString1$, OutString2$)
```

2. Then call the **SUB** procedure with the output arguments:

```
CALL AddString(A$,LEN(A$),B$,LEN(B$),C$,D$)
```

3. In the MASM data segment, eliminate `descriptor1` and add another output data block:

```
outbuffer2     db "It is a tale told by an idiot"
outbufferlen   dw $-outstring2
```

4. Add output parameters to the proc statement:

```
addstring proc s1:far ptr,s1len,s2:far ptr,s2len,s3:far ptr,s4:far ptr
```

5. Then transfer each fixed-string output to one of the passed-in variable-length strings using the **StringAssign** routine.

   For example:

```
push   ds                      ;push far pointer
lea    ax,outbuffer2           ;to fixed-length string.
push   ax
mov    ax, outbuffer2len
push   ax                      ;push length
les    ax,s3                   ;push far pointer to output
push   es                      ;string descriptor and 0 length
push   ax                      ;indicating that destination
xor    ax,ax                   ;is a variable-length string.
push   ax
call   stringassign            ;go transfer string data
```

MASM Calling BASIC

This example shows how to pass variable-length strings between MASM and BASIC using the general method explained in the preceding section. The first module is the MXSHKA.ASM file. For proper initialization, however, all mixed-language programs involving BASIC must start in BASIC. So startup begins in the second module, MXADSTB.BAS, which then calls the MASM procedure `shakespeare`.

The `shakespeare` procedure creates the strings `phrase1` and `phrase2`. For each string it also creates a data block that contains the length and a near pointer to the data. The elements of the data block are then passed to the BASIC procedure `AddString` by reference, along with a similar data block for output. The `AddString` procedure transfers the data to its own work area and then concatenates the two strings. It then transfers the output to a MASM fixed-length string sentence. Upon return, the MASM module prints the output string.

```
;***************************** SHAKESPEARE ********************************
; This program is found in file MXSHKA.ASM.
; It creates two strings and passes them to a BASIC procedure called
; AddString (in file MXADSTB.BAS).  This procedure concatenates
; the strings and passes the result to MASM which prints it.

.model medium,basic             ;Use same memory model as BASIC
.stack
.data

;Create the data
phrase1      db     "To be or not to be;"
phrase1len   dw     $-phrase1
phrase1off   dw     phrase1
phrase2      db     " that is the question."
phrase2len   dw     $-phrase2
phrase2off   dw     phrase2
sentence     db     100 dup(0)          ;Make room for return data
sentencelen  dw     0                   ;and a length indicator.
sentenceoff  dw     sentence

.code
shakespeare procuses si
```

```
;First call BASIC to concatenate strings:
        lea     ax,phrase1off           ;Push far address of
        push    ax                      ;fixed-length string #1,
        lea     ax,phrase1len           ;and its length.
        push    ax
        lea     ax,phrase2off           ;Do the same for the
        push    ax                      ;address of string #2,
        lea     ax,phrase2len           ;and its length.
        push    ax
        lea     ax,sentenceoff          ;Push far address of
        push    ax                      ;the return string,
        lea     ax,sentencelen          ;and its length.
        push    ax
        extrn   addstring:proc          ;Call BASIC function to
        call    addstring               ;concatenate the strings and
                                        ;put the result in the
                                        ;fixed-length return string.

; Call DOS to print string. The DOS string output routine (09H)
; requires that strings end with a "$" character.
        mov     bx,sentencelen          ;Go to end of the result
        lea     si,sentence             ;string and add a
        mov     byte ptr [bx + si],24h  ;"$" (24h) character.

        lea     dx,sentence             ;Set up registers
        mov     ah,9                    ;and  call DOS
        int     21h                     ;to print result string.
        ret

shakespeare endp

        end
```

```
'************************MXADSTRB.BAS****************************
DEFINT A-Z

'Start program in BASIC for proper initialization.
' Define external and internal procedures.
DECLARE SUB shakespeare ()
DECLARE SUB StringAssign(BYVAL srcsegment,BYVAL srcoffset,_
                         BYVAL srclen,BYVAL destsegment,_
                         BYVAL destoffset,BYVAL destlen)
DECLARE SUB addstring (instrg1off,instrg1len,instrg2off,_
                       instrg2len,outstrgoff,outstrglen)
DECLARE SUB StringRelease (s$)

'Go to main routine in second language
CALL shakespeare

'The non-BASIC program calls this SUB to add the two strings together
SUB addstring (instrg1off,instrg1len,instrg2off,instrg2len,_
               outstrgoff,outstrglen)

' Create variable-length strings and transfer non-BASIC fixed strings
' to them. Use VARSEG() to compute the segment of the strings
' returned from the other language--this is the DGROUP segment,
' and all string descriptors are found in this segment (even
' though the far string itself is elsewhere).

CALL StringAssign(VARSEG(a$), instrg1off, instrg1len, VARSEG(a$),_
                  VARPTR(a$), 0)
CALL StringAssign(VARSEG(b$), instrg2off, instrg2len, VARSEG(b$),_
                  VARPTR(b$), 0)

' Process the strings--in this case, add them.
c$ = a$ + b$

' Calculate the new output length.
outstrglen = LEN(c$)

' Transfer string output to a non-BASIC fixed-length string.
CALL StringAssign(VARSEG(c$), VARPTR(c$), 0, VARSEG(c$), outstrgoff,_
                  outstrglen)
END SUB
```

BASIC Calling C

This example shows how to pass variable-length strings between BASIC and C using the general method explained previously. The first module, MXSHKB.BAS, creates the strings `A$` and `B$`, then passes their string-descriptor far pointers and data lengths to a C procedure called `AddString`. This procedure is contained in the file MXADSTC.C. The `AddString` procedure transfers the data to its own work area and then concatenates the two strings. It then transfers the output to a BASIC variable-length string. Upon return, the BASIC module prints the output string.

```
'*******************************MXSHKB.BAS*******************************
DEFINT A-Z
'Define non-basic procedures
DECLARE FUNCTION addstring$(SEG s1$,BYVAL s1length,SEG s2$,BYVAL s2length)

'Create the data
A$ = "To be or not to be;"
B$ = " that is the question."
'Use Non-BASIC function to add two BASIC far strings.
C$ = addstring(A$, LEN(A$), B$, LEN(B$))

'Print the result on the screen.

PRINT C$
```

```
/*              MXADSTC.C                */
#include <string.h>

/* Function Prototypes force either correct data typing or compiler
 * warnings. Note all functions exported to BASIC and all BASIC (extern)
 * functions are declared with the far pascal calling convention.
 * WARNING: This must be compiled with the Medium memory model (/AM)
 */

char * pascal addstring( char far *s1, int s1len,
            char far *s2, int s2len );
extern void far pascal StringAssign( char far *source, int slen,
                    char far *dest, int dlen );

/* Declare global char array to contain new BASIC string descriptor.
 */
char BASICDesc[4];

char * pascal addstring( char far *s1, int s1len,
            char far *s2, int s2len )
```

```
{
    char TS1[50];
    char TS2[50];
    char TSBig[100];

    /* Use the BASIC StringAssign routine to retrieve information
     * from the descriptors, s1 and s2, and place them in the temporary
     * arrays TS1 and TS2.
     */
    StringAssign( s1, 0, TS1, 49 ); /* Get S1 as array of char */
    StringAssign( s2, 0, TS2, 49 ); /* Get S2 as array of char */

    /* Copy the data from TS1 into TSBig, then append the data from
     * TS2.
     */
    memcpy( TSBig, TS1, s1len );
    memcpy( &TSBig[s1len], TS2, s2len );

    StringAssign( TSBig, s1len + s2len, BASICDesc, 0 );

    return BASICDesc;
}
```

When returning to BASIC with only one string output, it is convenient for the C program to be a function, as in the preceding example. To pass back more than one string, do the following:

1. Have BASIC declare the procedure as a BASIC **SUB** procedure with output parameters included in the calling list:

```
DECLARE SUB AddString (SEG S1$,BYVAL S1Length,SEG S2$,BYVAL S2Length,_
                       OutString1$,OutString2$)
```

2. Call the **SUB** procedure with the output arguments `C$` and `D$`:

```
CALL AddString(A$, LEN(A$), B$, LEN(B$), C$, D$)
```

Suppose you want the original sentence returned in `C$` and its reverse in `D$`. Make these modifications to the C code:

1. Change the prototype and function header for `AddString` to include the outputs, and declare it a void.

```
void near * far pascal addstring( char far * s1, int s1len, char
far * s2, int s2len, char far * s3, char far * s4 );
```

2. Eliminate the data block for the string descriptor `BASICDesc[4]`. Change the **StringAssign** call so that `TSBig` is assigned to `s3` instead of `BASICDesc`.
3. Add the following lines of code:

```
TSBig[s1len + s2len] = '\0';
strrev( TSBig );
StringAssign( TSBig, s1len + s2len, s4, 0 );
```

4. Delete the **return** statement.

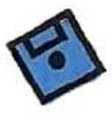
C Calling BASIC

This example shows how to pass variable-length strings between C and BASIC using the general method explained previously. The first module is the MXSHKC.C file. For proper initialization, however, all mixed-language programs involving BASIC must start in BASIC. So startup begins in a second module, MXADSTB.BAS, which then calls the C procedure `shakespeare`.

The `shakespeare` procedure creates the strings `s1` and `s2`. For each string it also creates a data block that contains the length and a near pointer to the data. The elements of the data block are then passed to the BASIC **SUB** procedure `AddString` by reference, along with a similar data block for output. The `AddString` **SUB** procedure transfers the data to its own work area and then concatenates the two strings. It then transfers the output to the C fixed-length string `s3`. Upon return, the C module prints the output string.

```
/*                          MXSHKC.C                          */
#include <stdio.h>
#include <string.h>

/* Prototype the shakespeare function (our function)
 * The prototypes force either correct data typing or compiler warnings.
 */
void far pascal shakespeare( void );
extern void far pascal addstring( char ** s1, int * s1len,
                                  char ** s2, int * s2len,
                                  char ** s3, int * s3len );

void far pascal shakespeare( void )
{
    char * s1 = "To be or not to be;";
    int  s1len;
    char * s2 = " that is the question.";
    int  s2len;
    char s3[100];
    int  s3len;
    char * s3ad = s3;
```

```
    s1len = strlen( s1 );
    s2len = strlen( s2 );
    addstring( &s1, &s1len, &s2, &s2len, &s3add, &s3len );

    s3[s3len] = '\0';
    printf("\n%s", s3 );
}
```



```
'*************************MXADSTRB.BAS*****************************
DEFINT A-Z

'Start program in BASIC for proper initialization.
' Define external and internal procedures.
DECLARE SUB shakespeare ()
DECLARE SUB StringAssign(BYVAL srcsegment,BYVAL srcoffset,_
                         BYVAL srclen,BYVAL destsegment,_
                         BYVAL destoffset,BYVAL destlen)
DECLARE SUB addstring (instrg1off,instrg1len,instrg2off,_
                       instrg2len,outstrgoff,outstrglen)
DECLARE SUB StringRelease (s$)

'Go to main routine in second language
CALL shakespeare

'The non-BASIC program calls this SUB to add the two strings together
SUB addstring (instrg1off,instrg1len,instrg2off,instrg2len,_
               outstrgoff,outstrglen)

' Create variable-length strings and transfer non-BASIC fixed strings
' to them. Use VARSEG() to compute the segment of the strings
' returned from the other language--this is the DGROUP segment,
' and all string descriptors are found in this segment (even
' though the far string itself is elsewhere).

CALL StringAssign(VARSEG(a$), instrg1off, instrg1len, VARSEG(a$),_
                  VARPTR(a$), 0)
CALL StringAssign(VARSEG(b$), instrg2off, instrg2len, VARSEG(b$),_
                  VARPTR(b$), 0)
```

```
' Process the strings--in this case, add them.
c$ = a$ + b$

' Calculate the new output length.
outstrglen = LEN(c$)
' Transfer string output to a non-BASIC fixed-length string.
CALL StringAssign(VARSEG(c$), VARPTR(c$), 0, VARSEG(c$), outstrgoff,_
                  outstrglen)
END SUB
```

BASIC Calling FORTRAN

This example shows how to pass variable-length strings between BASIC and FORTRAN using the general method explained above. The first module, MXSHKB.BAS, creates the strings `A$` and `B$`, then passes their string-descriptor far pointers and data lengths to a FORTRAN procedure called `ADDSTR`. This procedure is contained in the file MXADSTF.FOR. `ADDSTR` transfers the data to its own work area and then concatenates the two strings. It then transfers the output to a BASIC variable-length string. Upon return, the BASIC module prints the output string.

```
'*********************************MXSHKB.BAS*********************************

DEFINT A-Z
'Define non-BASIC procedures.
DECLARE FUNCTION AddString$(SEG S1$,BYVAL S1Length,SEG S2$,BYVAL S2Length)


'Create the data.
A$ = "To be or not to be;"
B$ = " that is the question."

'Use Non-BASIC function to add two BASIC far strings.
C$ = AddString(A$, LEN(A$), B$, LEN(B$))

'Print the result on the screen.
PRINT C$
```

```
C ********************* ADDSTRING  **********************
C This program is in file MXADSTF.FOR
C Declare interface to Stringassign subprogram. The pointer fields are
C declared INTEGER*4, so that different types of far pointers can be
C passed without conflict. The INTEGER*4 fields are essentially generic
C pointers. [VALUE] must be specified, or FORTRAN will pass pointers to
C pointers. INTEGER*2 also passed by [VALUE], to be consistent with
C declaration of StringAssign.
C
      INTERFACE TO SUBROUTINE STRASG [ALIAS:'STRINGASSIGN'] (S,SL,D,DL)
      INTEGER*4 S [VALUE]
      INTEGER*2 SL [VALUE]
      INTEGER*4 D [VALUE]
      INTEGER*2 DL [VALUE]
      END
C
C Declare heading of Addstring function in the same way as above: the
C pointer fields are INTEGER*4
C
      INTEGER*2 FUNCTION ADDSTR [ALIAS:'ADDSTRING'] (S1,S1LEN,S2,S2LEN)
      INTEGER*4 S1 [VALUE]
      INTEGER*2 S1LEN [VALUE]
      INTEGER*4 S2 [VALUE]
      INTEGER*2 S2LEN [VALUE]
C
C Local parameters TS1, TS2, and BIGSTR are temporary strings. STRDES is
C a four-byte object into which Stringassign will put BASIC string
C descriptor.
C
      CHARACTER*50 TS1, TS2
      CHARACTER*100 BIGSTR
      INTEGER*4 STRDES

      TS1 = " "
      TS2 = " "
      STRDES = 0
```

```
C
C Use the LOCFAR function to take the far address of data. LOCFAR returns
C a value of type INTEGER*4.
C
      CALL STRASG (S1, 0, LOCFAR(TS1), S1LEN)
      CALL STRASG (S2, 0, LOCFAR(TS2), S2LEN)
      BIGSTR = TS1(1:S1LEN) // TS2(1:S2LEN)
      CALL STRASG (LOCFAR(BIGSTR), S1LEN+S2LEN, LOCFAR(STRDES), 0)
      ADDSTR = LOC(STRDES)
      RETURN
      END
```

Instead of returning a string as the output of a function, you can also pass a string back as a subroutine parameter. In fact, to pass back more than one string, you must use this method. To do so, make these changes to the preceding code:

1. Declare the subroutine as a BASIC **SUB** procedure with the output parameter included in the calling list:

   ```
   DECLARE SUB AddString(SEG S1$,BYVAL S1Len SEG S2$,BYVAL S2Len,OutStr$)
   ```

2. Call the subprogram with output argument `C$`:

   ```
   CALL AddString(A$,LEN(A$),B$,LEN(B$),C$)
   ```

3. In the FORTRAN module, change the **FUNCTION** declaration to a **SUBROUTINE** declaration, in which the subroutine accepts an additional parameter not passed by value:

   ```
   SUBROUTINE ADDSTR[ALIAS:'ADDSTRING'](S1, S1LEN, S2, S2LEN, OUTS)
   INTEGER*4 S2 [VALUE]
   INTEGER*2 S2LEN [VALUE]
   INTEGER*4 S1 [VALUE]
   INTEGER*2 S1LEN [VALUE]
   INTEGER*4 OUTS
   ```

4. Change the line of code that sets the following return value:

   ```
   ADDSTR = LOC(STRDES)
   ```

   Into a statement that sets the value of the following output string:

   ```
   OUTS = STRDES
   ```

In some cases, you may want to return several strings. To do so, simply add additional `INTEGER*4` parameters to your procedure declaration.

FORTRAN Calling BASIC

This example shows how to pass variable-length strings between FORTRAN and BASIC using the general method explained previously. The first module is the MXSHKF.FOR file. For proper initialization, however, all mixed-language programs involving BASIC must start in BASIC. So startup begins in a second module, MXADSTB.BAS, which then calls the FORTRAN procedure `SHAKES`.

The `SHAKES` procedure creates the strings `STR1` and `STR2`. For each string it also creates a data block that contains the length and a near pointer to the data. The elements of the data block are then passed to the BASIC `AddString` procedure by reference, along with a similar data block for output. The `AddString` procedure transfers the data to its own work area and then concatenates the two strings. It then transfers the output to the FORTRAN fixed-length string `STR3`. Upon return, the FORTRAN module prints the output string.

```
C ********************** SHAKESPEARE ****************
C This program is in file MXSHKF.FOR
C Declare interface to BASIC routine ADDSTRING.
C All parameters must be passed NEAR, for compatibility with BASIC's
C conventions.
C


      INTERFACE TO SUBROUTINE ADDSTR[ALIAS:'ADDSTRING']
     * (S1,L1,S2,L2,S3,L3)
      INTEGER*2 S1 [NEAR]
      INTEGER*2 L1 [NEAR]
      INTEGER*2 S2 [NEAR]
      INTEGER*2 L2 [NEAR]
      INTEGER*2 S3 [NEAR]
      INTEGER*2 L3 [NEAR]
      END
C
C Declare subroutine SHAKESPEARE, which declares two strings, calls
C BASIC subroutine ADDSTRING, and prints the result.
C
      SUBROUTINE SHAKES [ALIAS:'SHAKESPEARE']
      CHARACTER*50 STR1, STR2
      CHARACTER*100 STR3
      INTEGER*2 STRLEN1, STRLEN2, STRLEN3
      INTEGER*2 TMP1, TMP2, TMP3
```

```
C
C The subroutine uses FORTRAN LEN_TRIM function, which returns the
C length of string, excluding trailing blanks. (All FORTRAN strings
C are initialized to blanks.)
C
      STR1 = 'To be or not to be;'
      STRLEN1 = LEN_TRIM(STR1)
      STR2 = ' that is the question.'
      STRLEN2 = LEN_TRIM(STR2)
      TMP1 = LOC(STR1)
      TMP2 = LOC(STR2)
      TMP3 = LOC(STR3)
      CALL ADDSTR (TMP1, STRLEN1, TMP2, STRLEN2, TMP3, STRLEN3)
      WRITE (*,*) STR3
END
```

```
'*************************MXADSTRB.BAS*****************************
DEFINT A-Z

'Start program in BASIC for proper initialization.
' Define external and internal procedures.
DECLARE SUB shakespeare ()
DECLARE SUB StringAssign(BYVAL srcsegment,BYVAL srcoffset,_
                         BYVAL srclen,BYVAL destsegment,_
                         BYVAL destoffset,BYVAL destlen)
DECLARE SUB addstring (instrg1off,instrg1len,instrg2off,_
                       instrg2len,outstrgoff,outstrglen)
DECLARE SUB StringRelease (s$)

'Go to main routine in second language
CALL shakespeare

'The non-BASIC program calls this SUB to add the two strings together
SUB addstring (instrg1off,instrg1len,instrg2off,instrg2len,_
               outstrgoff,outstrglen)

' Create variable-length strings and transfer non-BASIC fixed strings
' to them. Use VARSEG() to compute the segment of the strings
' returned from the other language--this is the DGROUP segment,
' and all string descriptors are found in this segment (even
' though the far string itself is elsewhere).

CALL StringAssign(VARSEG(a$), instrg1off, instrg1len, VARSEG(a$),_
                  VARPTR(a$), 0)
CALL StringAssign(VARSEG(b$), instrg2off, instrg2len, VARSEG(b$),_
                  VARPTR(b$), 0)
```

```
' Process the strings--in this case, add them.
c$ = a$ + b$

' Calculate the new output length.
outstrglen = LEN(c$)

' Transfer string output to a non-BASIC fixed-length string.
CALL StringAssign(VARSEG(c$), VARPTR(c$), 0, VARSEG(c$), outstrgoff,_
                  outstrglen)
END SUB
```

BASIC Calling Pascal

This example shows how to pass variable-length strings between BASIC and Pascal using the general method explained previously. The first module, MXSHKB.BAS, creates the strings `A$` and `B$`, then passes their string-descriptor far pointers and data lengths to a Pascal procedure called `ADDSTRING`. This procedure is contained in the file MXADSTP.PAS. `ADDSTRING` transfers the data to its own work area and then concatenates the two strings. It then transfers the output to a BASIC variable-length string. Upon return, the BASIC module prints the output string.

```
'*******************************MXSHKB.BAS*********************************
DEFINT A-Z
'Define non-basic procedures.
DECLARE FUNCTION AddString$(SEG S1$,BYVAL S1Length,SEG S2$,_
                            BYVAL S2Length)

'Create the data.
A$ = "To be or not to be;"
B$ = " that is the question."

'Use Non-BASIC function to add two BASIC far strings.
C$ = AddString(A$, LEN(A$), B$, LEN(B$))

'Print the result on the screen.
PRINT C$
```

```
{ **********************ADDSTRING ************************
  This program is in file MXADSTP.PAS  }

{ Module MXADSTP--takes address and lengths of two BASIC
  strings, concatenates, and creates a BASIC string descriptor. }

MODULE MAXADSTP;
{ Declare type ADSCHAR for all pointer types. For ease of programming,
  all address variables in this module are considered pointers to
  characters, and all strings and string descriptors are considered
  arrays of characters. Also, declare the BASIC string descriptor
  type as a simple array of four characters. }

TYPE
    ADSCHAR = ADS OF CHAR;
    ADRCHAR = ADR OF CHAR;
    STRDESC = ARRAY[0..3] OF CHAR;
VAR
    MYDESC : STRDESC;
{ Interface to procedure BASIC routine StringAssign. If source
  string is a fixed-length string, S points to string data and SL
  gives length. If source string is a BASIC variable-length string,
  S points to a BASIC string descriptor and SL is 0. Similarly for
  destination string, D and DL. }
PROCEDURE STRINGASSIGN (S:ADSCHAR; SL:INTEGER;
        D:ADSCHAR; DL:INTEGER ); EXTERN;

FUNCTION ADDSTRING (S1:ADSCHAR; S1LEN:INTEGER;
         S2:ADSCHAR; S2LEN:INTEGER) : ADRCHAR;

    VAR
   BIGSTR : ARRAY[0..99] OF CHAR;
{ Execute function by copying S1 to the array BIGSTR, appending S2
  to the end, and then copying combined data to the string descriptor. }

    BEGIN
   STRINGASSIGN (S1, 0, ADS BIGSTR[0], S1LEN);
   STRINGASSIGN (S2, 0, ADS BIGSTR[S1LEN], S2LEN);
   STRINGASSIGN (ADS BIGSTR[0], S1LEN+S2LEN, ADS MYDESC[0], 0);
   ADDSTRING := ADR MYDESC;
    END;  { End Addstring function,}
END.  {End module.}
```

Instead of returning a string as the output of a function, you can also pass a string back as a procedure parameter. In fact, to pass back more than one string, you must use this method. To do so, make the following changes to the preceding code:

1. Declare the procedure as a BASIC **SUB** procedure with the output parameter included in the calling list:

```
DECLARE SUB AddString(SEG S1$,BYVAL S1Len,SEG S2$,BYVAL S2Len,OutStr$)
```

2. Then call the subprogram with output argument `C$`:

```
CALL AddString (A$, LEN(A$), B$, LEN(B$), C$)
```

3. In the Pascal module, change the `EXTERN` declaration from a function to a procedure, in which the procedure accepts an additional `VAR` parameter:

```
PROCEDURE ADDSTRING (S1:ADSCHAR; S1LEN:INTEGER;
S2:ADSCHAR; S2LEN:INTEGER;
VAR OUTSTR:STRDESC); EXTERN;
```

4. Change the line of code that sets the following return value:

```
ADDSTRING:=MYDESC;
```

   Into a statement that sets the value of the following output string:

```
OUTSTR:=MYDESC;
```

In some cases, you may want to return several strings. To do so, simply add additional **VAR** parameters to your procedure declaration. Each formal argument (parameter) should be of type `STRDESC`. You can use another name for this type, but remember to define this type or some equivalent type `(ARRAY[0..3] OF CHAR)` at the beginning of your Pascal module.

Pascal Calling BASIC

This example shows how to pass variable-length strings between Pascal and BASIC using the general method explained previously. The first module is the MXSHKP.PAS file. For proper initialization, however, all mixed-language programs involving BASIC must start in BASIC. So startup begins in a second module, MXADSTB.BAS, which then calls the Pascal procedure `SHAKESPEARE`.

The `SHAKESPEARE` procedure creates the strings `STR1` and `STR2`. For each string it also creates a data block that contains the length and a near pointer to the data. The elements of the data block are then passed to the BASIC procedure `AddString` by reference, along with a similar data block for output. The `AddString` procedure transfers the data to its own work area and then concatenates the two strings. It then transfers the output to the Pascal fixed-length string `STR3`. Upon return, the Pascal module prints the output string.

```
{ ************************ SHAKESPEARE ******************
  This program is in file MXSHKP.PAS }

MODULE MPAS;
TYPE
    ADRCHAR = ADR OF CHAR;
VAR
    S1, S2, S3 : LSTRING (100);
    S1LEN, S2LEN, S3LEN : INTEGER;
    TMP1, TMP2, TMP3 : ADRCHAR;
{ Declare interface to procedure ADDSTRING, which concatenates first
  two strings passed and places the result in the third string
  passed. }
PROCEDURE ADDSTRING (VAR TMP1:ADRCHAR; VAR STR1LEN:INTEGER;
          VAR TMP2:ADRCHAR; VAR STR2LEN:INTEGER;
          VAR TMP3:ADRCHAR; VAR STR3LEN:INTEGER ); EXTERN;

{ Procedure Shakespeare declares two strings, calls BASIC procedure
  AddString to concatenate them, then prints results. With LSTRING
  type, note that element 0 contains length byte. String data starts
  with element 1. }
PROCEDURE SHAKESPEARE;
    BEGIN
   S1:='To be or not to be;';
   S1LEN:=ORD(S1[0]);
   S2:=' that is the question.';
   S2LEN:=ORD(S2[0]);
   TMP1:=ADR(S1[1]);
   TMP2:=ADR(S2[1]);
   TMP3:=ADR(S3[1]);
   ADDSTRING (TMP1, S1LEN, TMP2, S2LEN, TMP3, S3LEN);
   S3[0]:=CHR(S3LEN);
   WRITELN(S3);
    END;
END.
```

```
'************************MXADSTRB.BAS****************************
DEFINT A-Z

'Start program in BASIC for proper initialization.
' Define external and internal procedures.
DECLARE SUB shakespeare ()
DECLARE SUB StringAssign(BYVAL srcsegment,BYVAL srcoffset,_
                         BYVAL srclen,BYVAL destsegment,_
                         BYVAL destoffset,BYVAL destlen)
DECLARE SUB addstring (instrg1off,instrg1len,instrg2off,_
                       instrg2len,outstrgoff,outstrglen)
DECLARE SUB StringRelease (s$)

'Go to main routine in second language
CALL shakespeare

'The non-BASIC program calls this SUB to add the two strings together
SUB addstring (instrg1off,instrg1len,instrg2off,instrg2len,_
               outstrgoff,outstrglen)

' Create variable-length strings and transfer non-BASIC fixed strings
' to them. Use VARSEG() to compute the segment of the strings
' returned from the other language--this is the DGROUP segment,
' and all string descriptors are found in this segment (even
' though the far string itself is elsewhere).

CALL StringAssign(VARSEG(a$), instrg1off, instrg1len, VARSEG(a$),_
                  VARPTR(a$), 0)
CALL StringAssign(VARSEG(b$), instrg2off, instrg2len, VARSEG(b$),_
                  VARPTR(b$), 0)

' Process the strings--in this case, add them.
c$ = a$ + b$

' Calculate the new output length.
outstrglen = LEN(c$)

' Transfer string output to a non-BASIC fixed-length string.
CALL StringAssign(VARSEG(c$), VARPTR(c$), 0, VARSEG(c$), outstrgoff,_
                  outstrglen)
END SUB
```

Passing Strings in QBX

If you have a BASIC module that calls an external procedure, as in the preceding examples, you may want to debug the BASIC code in QBX. If you are using any of the string-processing routines, you'll need to make a Quick library that contains the routines. Here's how to do this:

1. Compile the external procedure.
2. Link the object file with the necessary libraries plus the QBXQLB.LIB library. Use the /Q option. For instance, for the first C example:

```
LINK MXADSTC.OBJ,,,CLIBC QBXQLB.LIB /Q
```

3. Load the new Quick library when starting up QBX by using the /L option as shown here:

```
QBX /L MXADSTC.QLB
```

If you just want to test the new string routines in QBX, without calling an external procedure, load the QBX.QLB Quick library when starting QBX.

Passing Variable-Length String Arrays

To manage variable-length string arrays, BASIC creates a data block in DGROUP containing a series of variable-length string descriptors—one for each array element. The descriptors begin at address of the first element in the array, for example the first descriptor in a three dimensional zero-based array named `A$()` would be `A$(0,0,0)`. The descriptors are in column-major order, where the rightmost dimension changes first. Note, however, that if you compile with the /R option the descriptors will be in row-major form. For examples of this, see the *Microsoft Mixed-Language Programming Guide*.

Using the **StringAssign** routine, a non-BASIC language can copy a BASIC variable-length string array into its own workspace, modify any data element (even change its length), and copy the changed array back to BASIC.

Assume, for example, that `A$()` is a one-dimensional BASIC string array that contains these elements indexed with numbers 1 to 10:

```
A,BB,CCC,DDDD,EEEEE,FFFFFF,GGGGGGG,HHHHHHHH,IIIIIIIII,JJJJJJJJJ
```

The elements need to be changed to:

```
jjjjjjjjj,iiiiiiiii,hhhhhhhh,ggggggg,ffffff,eeeee,dddd,ccc,bb,a
```

To accomplish this, BASIC could call a MASM procedure, passing it the address of the first string descriptor in the array:

```
DECLARE SUB ChangeArray(S$)
CALL ChangeArray(A$(1))
```

The array transfer is accomplished by:

```
        .model medium,basic

        .data
array           dw  100 dup(0)             ;Create space for 10 element array
        .code
changearray   proc uses si di, arraydescriptor: near ptr ;pointer to array
        extrn   stringassign:proc          ;declare BASIC callback
        mov     cx, 10                     ;number of transfers
        mov     si, arraydescriptor        ;first source
        lea     di, array                  ;first destination
transferin:  push   cx                     ;preserve cx during callback
        push    ds                         ;far pointer to source--
        push    si
        xor     ax,ax                      ;a variable-length string
        push    ax
        push    ds                         ;far pointer to destination--
        push    di                         ;a fixed-length string
        mov     ax, 10                     ;10 bytes long
        push    ax
        call    stringassign               ;go transfer one string
        pop     cx                         ;restore cx
        add     si, 4                      ;update pointers
        add     di,10
        loop    transferin                 ;last transfer?

;Now, change the data to lower case

        mov     cx,100
        lea     bx, array
more:   cmpbyte ptr[bx], 0
        jz      skip
        addbyte ptr[bx], 32
skip:   inc     bx
        loop    more
```

```
; and send it back out, last element first.

        mov     cx, 10                  ;number of transfers
        lea     si, array + 90          ;first source--the last element
        mov     di, arraydescriptor     ;first destination
transferout: push   cx                  ;preserve cx during call

        push    ds                      ;far pointer to source--
        push    si
        push    cx                      ;a fixed-length string
        push    ds                      ;far pointer to destination--
        push    di                      ;a variable-length
        xor     ax,ax                   ;string.
        push    ax
        call    stringassign            ;go transfer one string
        pop     cx                      ;restore variables
        sub     si, 10                  ;update pointers
        add     di, 4
        loop    transferout             ;last transfer?

        ret

changearray  endp
        end
```

Passing Fixed-Length Strings

If you want to pass BASIC fixed-length string data to and from another language, you can also use the **StringAssign** routine. This works in spite of the fact that fixed-length strings are illegal as parameters in BASIC procedures. In other words, this is impossible:

```
DECLARE FUNCTION PassFixed AS STRING * 10 (FixedString1 AS STRING * 10)
```

You can, however, achieve the same result by using fixed-string arguments and variable-string parameters:

```
' Declare an external function.
DECLARE FUNCTION PassFixed$(FixedString1$)
DIM InputString AS STRING * 10, OutputString AS STRING * 20
OutputString = PassFixed(InputString)
```

When BASIC makes the call, it creates the variable-length string `FixedString1$` and copies the data from the fixed string into it. It pushes the address of the `FixedString1$` descriptor (remember, this is a variable-length string) onto the stack.

From here on, processing is the same as for the preceding examples. The called function, before returning, creates a 4-byte string descriptor filled with zeros. It uses **StringAssign** to transfer its output data to this newly created variable-length string. The address of the string's descriptor is placed in AX and the return is made.

When the equal operator (=) is executed in the last code statement, BASIC assigns the data from the returned variable-length string to the the fixed-length string `OutputString`.

Getting Online Help

Online Help is available in QBX for mixed-language programming. For help with making an external call, see the **CALL**, **CALLS**, and **DECLARE** non-BASIC Statement categories in the Help Keyword Index. More information can be found in the Mixed-Language Programming section listed in the Help Table of Contents.

Sample code in the online Help screens can be copied and pasted to a file. All copied BASIC code samples will execute within QBX. Any of the code samples can also be compiled, linked into executable files or Quick libraries, or made into object-module libraries. See Chapters 18, "Using LINK and LIB" and 19, "Creating and Using Quick Libraries," for more information.

Chapter 14
OS/2 Programming

Microsoft BASIC enables you to create programs for the OS/2 protected-mode environment as well as the real-mode (DOS) environment. This chapter explains how OS/2 protected-mode programs differ from BASIC programs written for DOS. You'll learn how to write, compile, and link programs that run under OS/2, as well as the following:

- What libraries and include files you'll need.
- How to call OS/2 functions in your program.
- Limitations for BASIC programs.
- Language changes for protected-mode-only programs.
- How to prepare your programs for debugging.

Creating Real or Protected-Mode Programs

With BASIC you can create protected-mode programs or real-mode programs. You cannot create bound programs—programs that run in real and protected modes. You also cannot bind a BASIC executable file after linking.

To create an OS/2 program, BASIC provides the QuickBASIC Extended (QBX) programming environment.

QBX can be run only in real mode, although you can create OS/2 programs that run in real or protected mode from QBX. The QBX environment contains context-sensitive online Help.

While this chapter assumes you are writing, compiling, and linking from QBX you can, if you wish, invoke the BASIC Compiler (BC) and the LINK utility separately from the command line.

Editing Source Code

To edit source code, you can use the editing facilities built into QBX. Use the F1 key to access online Help for information about using specific editing features and commands.

Except where noted, you may write your program using the BASIC statements and functions described in the *BASIC Language Reference*. There are certain BASIC statements and functions, however, that behave differently or require extra caution in protected mode. These are described in the following sections.

Language Changes for Protected Mode

This section describes specific BASIC statements and functions that behave differently or require extra caution in protected mode. The most significant differences between real mode and protected mode concern memory management. Others concern BASIC statements that directly access the machine's hardware, an activity that is not always appropriate in a protected environment. Table 14.1 lists and explains the statements and functions that are changed for protected mode in Microsoft BASIC.

Table 14.1 Changes to BASIC Statements and Functions for Protected Mode

Keywords	Use in protected mode
BEGINTRANS, BOF, CHECKPOINT, CLOSE, COMMITTRANS, CREATEINDEX, DELETE*element*, EOF, FILEATTR, GETINDEX$, INSERT, LOF, MOVE, OPEN, RETRIEVE, ROLLBACK, SAVEPOINT, SETINDEX, TYPE, UPDATE	None of these keywords are supported for use with ISAM. ISAM is not supported in protected mode.
BLOAD	You must be able to write to the target of any **BLOAD** operation from the executing process. If you attempt to refer to a memory address for which your process does not have write permission, BASIC may generate the error message `Permission denied`, or the operating system may generate a protection exception.
BSAVE	The target of any **BSAVE** operation must be readable from the executing process. If you attempt to refer to a memory address for which your process does not have read permission, BASIC may generate the error message `Permission denied`, or the operating system may generate a protection exception.
CALL	You can use the **CALL** statement, along with the appropriate **DECLARE** statements, to call routines directly in dynamic-link libraries or OS/2 functions. The preferred method for calling OS/2 functions is to include the file DOSCALLS.BI, which contains function declarations and data-structure definitions for all OS/2 routines. For more information about using DOSCALLS.BI, see "Calling OS/2 Functions" later in this chapter.

Table 14.1 Continued

Keywords	Use in protected mode
CALL ABSOLUTE	The target of any **CALL ABSOLUTE** statement must lie within memory that your program can access. Before it executes **CALL ABSOLUTE**, BASIC attempts to get an executable code segment alias for the code you wish to access. If this operation fails, BASIC generates the error message `Permission denied`. A safe place to store user-written machine code is in memory allocated for a conventional BASIC object, such as an array.
CALL INT86, CALL INT86X, CALL INT86OLD, CALL INTERRUPT	You may not use the **CALL INT86, CALL INT86X, CALL INT86OLD**, or **CALL INTERRUPT** statements. You should replace such statements with equivalent OS/2 function invocations.
COLOR	BASIC ignores **COLOR** statements in screen mode 1.
DEF SEG	Any **DEF SEG** statement must refer only to a valid selector. The **DEF SEG** statement itself does not generate any memory references using the selector, nor does it attempt to validate the selector. If a misdirected **DEF SEG** statement causes your program to refer to an illegal memory address, the operating system may generate a protection exception, or BASIC may generate the `Permission denied` error message. The default **DEF SEG** segment always constitutes a valid memory reference. Use caution when altering this reference in protected mode.
INP	The **INP** function is not available.
IOCTL, IOCTL$	The **IOCTL** statement and **IOCTL$** function, which communicate directly with device drivers, are not available. However, you can achieve the same effect by directly invoking OS/2 functions.
ON *event*	You cannot use the **ON PEN, ON PLAY**, or **ON STRIG** statements.
OUT	The **OUT** statement is not available.
PALETTE, PALETTE USING	The **PALETTE** and **PALETTE USING** statements are not available.
PEEK	Any address referred to by the **PEEK** function must be readable. If a **PEEK** function refers to an address for which your process does not have read permission, the operating system may generate a protection exception, or BASIC may generate the error message `Permission denied`.

Table 14.1 *Continued*

Keywords	Use in protected mode
PEN	You cannot use the **PEN** function, or the **PEN ON**, **PEN OFF**, or **PEN STOP** statements.
PLAY	The **PLAY** statement is not available.
POKE	You must be able to write to any address referred to by a **POKE** statement. If a **POKE** statement refers to a memory address for which your process does not have write permission, the operating system may generate a protection exception, or BASIC may generate the error message `Permission denied`.
SCREEN	Screen modes 0-2 are available in protected mode. In real mode, BASIC supports screen modes 0-3 and 7-13. BASIC does not support multiple screen pages in protected mode (**SCREEN** ignores the active page and visual page parameters).
SETMEM	The **SETMEM** function performs no function other than to return a dummy value. This value is the previous **SETMEM** value adjusted by the **SETMEM** parameter. The initial value for **SETMEM** is 655,360.
SOUND	The **SOUND** statement is not available.
STICK	The **STICK** function is not available.
STRIG	You cannot use the **STRIG** function, or the **STRIG ON**, **STRIG OFF**, or **STRIG STOP** statements.
VARSEG	The **VARSEG** function returns the selector of the specified variable or array.
WAIT	The **WAIT** statement is not available.

Protected-Mode-Only Statements

The following statements are supported only while running in the protected mode of OS/2:

Statement	Description
ON SIGNAL	Defines an event trap for a protect-mode signal.
OPEN PIPE	Executes another process whose input and output are connected to your BASIC program.
SHELL	Executes another process without suspending your BASIC program.

For more details about these statements, see the *BASIC Language Reference*.

Making OS/2 Calls in Your Program

Microsoft BASIC allows you to make direct calls to OS/2 functions when running in protected mode. When invoking an OS/2 function, it is necessary to use the syntax for calling a BASIC function, even if you are not interested in the error code or other information that might be returned by the OS/2 function.

For example, the following program fragment invokes the OS/2 function **DOSBEEP**:

```
' Include the file BSEDOSPC.BI
REM $INCLUDE: 'bsedospc.bi'
' Invoke the DOSBEEP function
x = DOSBEEP(100, 200)
```

OS/2 Include Files

To provide support for both OS/2 function calls and type definitions for data structures used by OS/2 functions, Microsoft BASIC provides several OS/2 include files. You can insert include files into your program with a **$INCLUDE** metacommand.

Microsoft BASIC provides the following OS/2 include files:

Include filename	Functions supported
BSEDOSPC.BI	Process control, semaphores, signals, pipes and queues, error handling and messages, session manager
BSEDOSFL.BI	Device drivers, file management
BSESUBMO.BI	Mouse
BSEDOSPE.BI	National language, resource management, module management, date/time and timer, memory management, information segments

These include files provide support only for OS/2 functions that are appropriate for BASIC. Many of the omitted functions involve multiple thread capabilities, while others duplicate existing BASIC support, such as keyboard and screen I/O. Because these include files provide a standard interface to OS/2 functions, you should avoid changing their contents unnecessarily.

The process of calling OS/2 functions requires certain data types that are not intrinsic to BASIC. Because of this, the include files listed in the preceding table use standard methods for simulating those types. How various data types are simulated in the OS/2 include files for BASIC is described in the following sections.

Unsigned Values

Unsigned values are not intrinsic to BASIC. The signed version of the given type is used instead.

Pointers in User-Defined Types

In cases where OS/2 requires a pointer as a field of a user-defined type, the include files use the type `ADDRESS`, which is defined at the beginning of the include file BSEDOSPC.BI. You can fill in the fields of this type with **VARSEG** and **SADD** in the case of a variable-length string, or **VARSEG** and **VARPTR** in the case of other data objects.

Far Character Pointers in Function Parameters

The far character pointer for OS/2 functions (far char *) is simulated with two parameters: a segment and an offset. Note that both of these values are integers. Thus, if the original declaration would have been DOSXYZ(far char *), it is declared in this form:

```
DOSXYZ( BYVAL S1s AS INTEGER, BYVAL S1o AS INTEGER )
```

You would call this function with a statement in the following form, where `FixedLen` is a fixed-length string:

```
DOSXYZ( VARSEG(FixedLen), VARPTR(FixedLen) )
```

You can call the function with a statement in the following form if `VarLen` is a variable-length string:

```
DOSXYZ( VARSEG(VarLen), SADD(VarLen) )
```

Pointer to a Function in a Function Parameter

The pointer to a function is simulated with two parameters, just as in the case of a far character pointer. The first integer represents the segment and the second integer represents the offset (see the preceding section). BASIC itself has no means of finding the location of a function; however, if you find that location with a language procedure written in another language, the address can be used in an OS/2 function call from BASIC.

Character in a Function Parameter

A character in a function parameter is simulated with an integer. This method is safe because parameters are always passed as words. Thus, a character is extended to an integer before being passed in other languages.

Example The following example prompts you for a file or set of files you want information on, and then uses the **DosFindFirst** and **DosFindNext** API functions to retrieve the information.

```
' This is an OS/2 protect mode program: compile with the /Lp switch

CONST TRUE = -1
CONST FALSE = 0
' $INCLUDE: 'BSEDOSFL.BI'

DEFINT A-Z

COLOR 15, 1
DIM buffer AS FILEFINDBUF
DIM Filelist(255) AS FILEFINDBUF
DIM reserved  AS LONG

CLS

PRINT "Test of DOSFINDFIRST..."

DO
   PRINT
   INPUT "Enter the Filename(s) : "; flname$
   flname$ = flname$ + CHR$(0)

   counter = 0
   atr = 0 + 2 + 4 + 16  'normal + hidden + system + subdirectory
   dirh = 1
   searchcount = 255
   bufflen = 36
   X = DosFindFirst%(VARSEG(flname$) SADD(flname$),dirh,_
                     atr,buffer,bufflen,searchcount,reserved)
   IF (X = 0) THEN
      DO
         counter = counter + 1
         Filelist(counter) = buffer

         ' clear out buffer for call to DosFindNext
         buffer.achName = STRING$(13, 32) 'assign blanks
         buffer.fdateLastWrite = 0
         buffer.ftimeLastWrite = 0
      LOOP WHILE (DosFindNext%(dirh, buffer, bufflen, searchcount) = 0)
ELSE
   PRINT "No MATCH was found"
   END
END IF
```

```
    PRINT : PRINT counter; " matching files found:": PRINT
    FOR t = 1 TO counter
        PRINT USING "###"; t; SPC(2);
        PRINT Filelist(t).achName
    NEXT t

    PRINT : INPUT "Repeat (y/n)"; y$

LOOP WHILE UCASE$(LEFT$(y$, 1)) = "Y"
```

Note This program should be compiled using near strings; to use far strings, use the **SSEG** and **SADD** functions instead to return the segment and offset of the filename you want. For more information about compiling programs that use far strings, see Chapter 11, "Advanced String Storage."

Creating Dynamic-Link Libraries

You cannot create dynamic-link libraries from BASIC modules created with Microsoft BASIC, but a protected-mode BASIC program can invoke routines contained in a dynamic-link library.

BASIC run-time and extended-run-time modules can be dynamic-link libraries. And user-created routines embedded in a protected-mode, extended run-time module are dynamically linked.

Creating Multiple Threads

A protected-mode BASIC program can call an external routine or execute a process that creates multiple threads. However, because Microsoft BASIC run-time routines are not re-entrant, you cannot create multiple threads from within a protected-mode BASIC program. For the same reason, threads created by external routines cannot call BASIC routines.

Making Memory References

Several BASIC statements and functions refer directly to addresses in the machine's physical memory. These include **CALL ABSOLUTE, DEF SEG, PEEK, POKE, BLOAD, BSAVE, VARPTR**, and **VARSEG**.

When using these statements in protected mode, you should take care not to refer to an illegal memory address. The selector portion of the address in question must refer to a memory segment for which your process has appropriate read and/or write permission. In addition, the segment must be large enough to contain the address referenced by the offset and the size of the object being accessed.

If your program ignores these requirements, it may trigger a protection exception by the operating system or the BASIC error message `Permission denied`.

The default **DEF SEG** segment is safely addressable. Values returned by **VARPTR** and **VARSEG** are valid; and you can safely access BASIC variables and arrays, provided you do not exceed the size of valid BASIC objects.

Using Graphics

In protected mode, all graphics operations are limited to BASIC screen modes 1 and 2. Screen modes 3 and 7–13 are supported only in real mode. Microsoft BASIC does not support multiple screen pages in protected mode.

Using Music, Sound, and Devices

Except for the **BEEP** statement, no music or sound statements are available in protected mode (you cannot use **SOUND** or **PLAY**). However, you can call the OS/2 function **DOSBEEP**.

You cannot use the light pen, joystick, and joystick triggers in protected mode.

Creating Extended Run-Time Modules

You can create extended run-time modules that can be used in real and protected mode. The program BUILDRTM.EXE is a bound program, so it can be run in either real or protected mode. As with the compiler, BUILDRTM creates an extended run-time module suitable for the environment you are in at the time. You can override the default environment by specifying either the /LP (protected mode) or /LR (real mode) compiler option in QBX.

To avoid errors, it is important that all modules in a given application share the same target environment (real or protected mode).

For information about the standard run-time modules and libraries, see "Using Standard Run-Time Modules and Libraries" later in this chapter.

Compiling OS/2 Programs

After writing your BASIC program, you can compile it from QBX. From QBX, choose Make EXE from the Run menu. For protected-mode programs, make sure to specify the /LP option to LINK.

Unless you specify otherwise, the compiler creates an object file suitable for the environment in which it was created. If you run the compiler under DOS or real mode, it automatically creates an object file suitable for DOS and OS/2 real mode. Likewise, if you run the compiler in protected mode, it creates a protected-mode object file.

You can override the default environment by using one of the options listed in Table 14.2.

Table 14.2 Environment Options

Option	Effect
/Lp	Creates protected-mode object file.
/Lr	Creates real-mode object file.

If you supply the /Lp option, the compiler creates a protected-mode object file no matter which environment you are in at the time. If you supply the /Lr option, the compiler creates a real-mode (DOS-compatible) object file.

Linking OS/2 Programs

Before linking OS/2 programs, you should read Chapter 18, "Using LINK and LIB." That chapter contains information about module definition files and import libraries, which are needed if your program makes calls to dynamic-link libraries.

From the QBX environment, your program is automatically linked with the proper libraries when you choose the Make EXE option from the Run menu. By default, LINK uses the libraries created by the BASIC Setup program. These libraries use the following naming convention:

BCL70*float string mode*.LIB

The possible values for each variable are as follows:

Variable	Letter	Feature
float	E	Emulator (/FPi)
	A	Alternate math (/FPa)
string	N	Near strings
	F	Far strings (/Fs)
mode	R	Real mode(/LR)
	P	Protected mode (/LP)

For example, if you specified the emulator floating-point option, near strings, and real-mode-only options during setup, your program would link, by default, with the BCL70ENR.LIB library. From QBX, you can change the default library by changing options in the Create EXE dialog box.

For information about LINK options you can use, see Chapter 18, "Using LINK and LIB."

Using Standard Run-Time Modules and Libraries

The BASIC Setup program automatically creates a run-time module and run-time module library that match the operating environment and floating-point method you specify. Run-time modules use the following naming convention:

File	Convention
Real-mode run-time module	BRT70*float string* R.EXE
Protected-mode run-time module	BRT70*float string* P.DLL
Run-time library	BRT70*float string mode*.LIB

The possible values for each variable are the same as shown in the preceding section.

If you create programs for more than one operating mode, you must make sure that the appropriate run-time module library is available when linking, and the appropriate run-time module is available at run time. This can be done by setting the LIB and PATH environment variables to the directories where your libraries and run time modules are kept, or by moving them into appropriate directories where they can be found when run.

LINK Options for Real and Protected Modes

The following options for the LINK utility can only be used when linking real-mode programs:

/CP[[ARMAXALLOC]]

/DS[[ALLOCATE]]

/HI[[GH]]

/NOG[[ROUPASSOCIATION]]

/O[[VERLAYINTERRUPT]]

The following options can only be used when linking protected-mode programs:

/A[[LIGNMENT]]:*size*

/W[[ARNFIXUP]]

For descriptions of these and other LINK options, see Chapter 18, "Using LINK and LIB."

Debugging OS/2 Programs

Microsoft BASIC provides the CodeView debugger to help you debug your OS/2 program. Two versions of CodeView are supplied: CVP.EXE, for debugging under protected mode and CV.EXE, for debugging under real mode.

To prepare files for use with CodeView, you must specify the /Zi option of the compiler and the /CO option of LINK. From QBX, choose the CodeView Information option in the Make EXE File dialog box.

Running BASIC Programs Under OS/2

When you run a BASIC program in OS/2 protected mode, the system needs to find one or more dynamic-link libraries (.DLL). If a needed dynamic-link library cannot be found at run time, the system displays an error message. When searching for a dynamic-link library, OS/2 looks in the directories specified by the LIBPATH configuration command in your CONFIG.SYS file. For more about OS/2 configuration commands, see Part 2 in the *Microsoft Operating System/2 User's Guide*.

In OS/2 protected mode, it is possible to run a BASIC program as a detached program (using the **DETACH** command). A detached program, however, cannot perform console input or output while it is detached. If input or output is attempted (including key trapping), a `Device unavailable` error message is generated. If a detached program causes an error, the error message appears in an OS/2 pop-up window.

See the *Microsoft Operating System/2 User's Guide* for more information about the **DETACH** command.

Chapter 15
Optimizing Program Size and Speed

Improved code generation and BASIC run-time management as well as higher-capacity development tools make it possible to write substantially larger BASIC programs, that can then be compiled into smaller and faster executables than previously possible.

This chapter consists of programming hints and technical information to help you write faster, more efficient BASIC programs. The first half focuses on making the most of available random-access memory (RAM) and ways to reduce the size of programs in memory and on disk. The second half of this chapter, beginning with the section "Compiling Programs for Speed," is a series of tips to help you improve the performance of BASIC programs.

Size and Capacity

While there is no substitute for good program design, new features like far strings, overlays, stub files, and improved code generation can help significantly reduce executable size and increase data capacity.

There are three major areas of BASIC size and capacity issues: BASIC variable space, BASIC program size, and the size of the compiled executable file on disk. An understanding of how BASIC programs allocate and use memory will help you to manage variable and program space to your advantage.

BASIC Memory Use

Under the DOS and OS/2 operating systems, RAM stores the resident portion of any BASIC program that is currently running, the various constants and data needed by the program, the variable space, and any other information needed by the computer while the program is running.

RAM used by a BASIC program is divided into two categories: near memory and far memory. Near memory and far memory each contains a "heap," which is an area of memory used to store dynamic variables. "Near memory," also referred to as "DGROUP," is the single segment of memory (maximum size of 64K) that includes, but is not limited to, the near heap (where near strings and variables are stored), the stack, and state information about the BASIC run-time. "Far memory" is the multiple-segment area of memory outside of DGROUP that includes, but is not limited to, the BASIC program (run-time and generated code) and the far heap (where dynamic arrays and far strings are stored).

DOS and OS/2 manage memory in fundamentally different ways. In DOS, application programs use physical addresses to directly access specific memory locations. In contrast, OS/2 supports virtual addressing. "Virtual addressing" is an advanced system of memory management whereby the operating system maintains an address space larger than physical memory by swapping data to and from disk (and within memory) as needed, while mapping address references in application programs onto the physical addresses of the actual memory locations. Under DOS, the upper boundary of standard memory is 640K. Under OS/2, program size is realistically limited only by available disk space.

The memory map in Figure A.1 shows the high-level memory organization for a BASIC program. The diagram shows the different areas of memory in their correct relative positions for a stand-alone (compiled with /O) BASIC program running under DOS or OS/2 real mode. A program compiled without /O under DOS loads the run-time module in far memory (above DGROUP), not next to the generated code as pictured in Figure 15.1. Also, all references to strings in the diagram refer only to variable-length strings, since fixed-length strings are managed in memory the same way fixed-length data types (such as numerics) are managed.

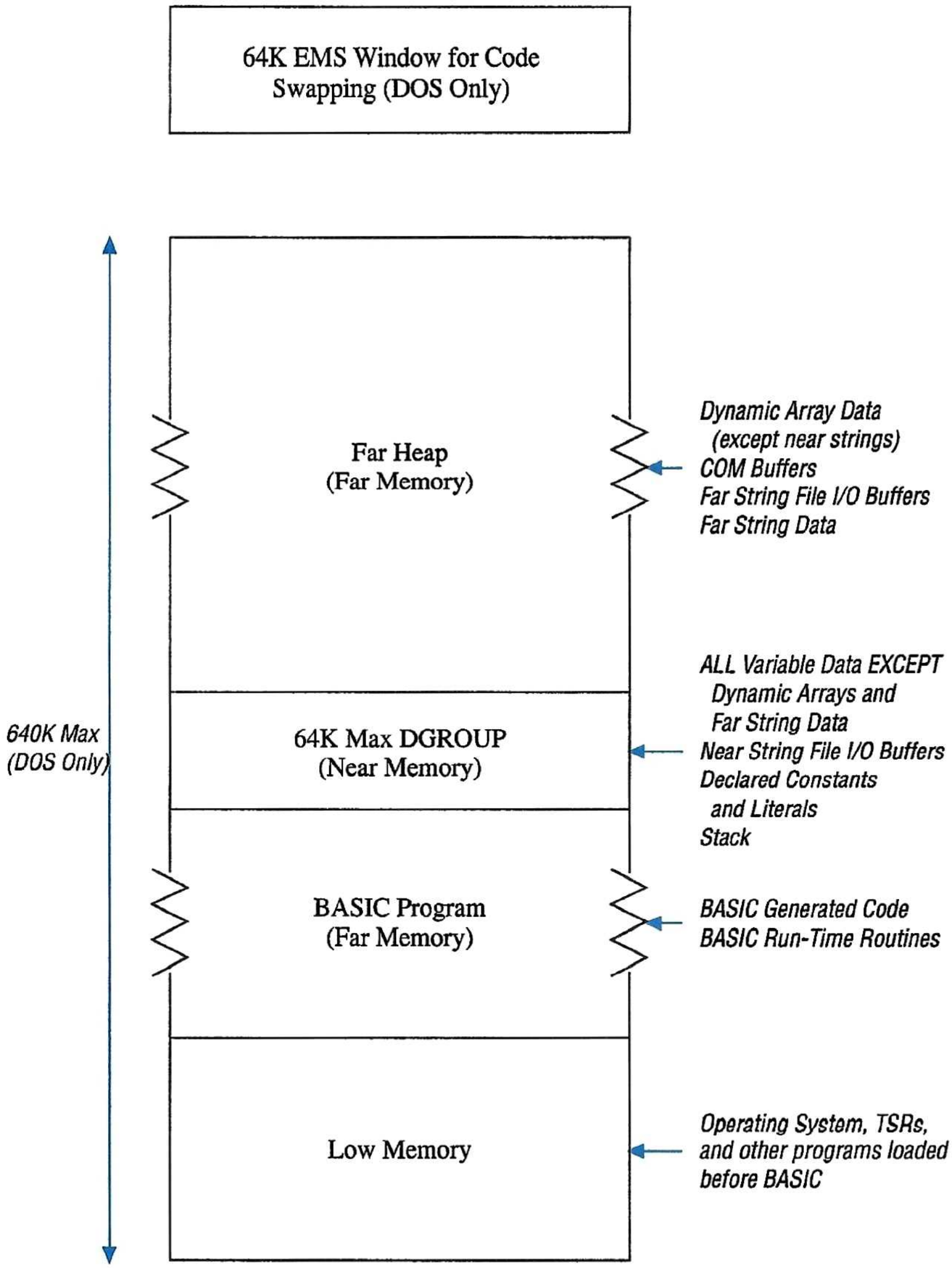


Figure 15.1 BASIC Memory Map

There are a few fundamental things you can do to maximize capacity in BASIC under DOS. If you are working with a large amount of code in a program, or if you want more far memory to be available for far string or dynamic array data, overlays will allow different module groups in your program to share the same memory space. Microsoft BASIC supports up to 64 overlays.

Each overlay may be up to 256K for a maximum total of over 15 megabytes of compiled code. See the section "Overlays" later in this chapter and Chapter 18, "Using LINK and LIB," for more information about overlays.

If DGROUP (near memory) is constraining, compiling with the far string option (/Fs) will place variable-length string data in the far heap, leaving more room in DGROUP for other data. See Chapter 11, "Advanced String Storage," for more information on using strings.

The **FRE()** function can give you information about exactly how much memory is available in a particular area of memory. `FRE(-2)` returns the amount of stack space not yet used. `FRE(-1)` returns the amount of available far memory. **FRE()** can also be used to determine the amount of string space available for near and far strings. See the *BASIC Language Reference* or online Help for a complete description of the **FRE()** function.

Variable-Length String Storage

Microsoft BASIC gives you two options for storing variable-length string data and string array data: near string storage in DGROUP and far string storage in the far heap. You can specify which option you want to use by compiling with or without the far string option (/Fs). The only data type affected by the /Fs option is the variable-length string data type. Table 15.1 shows where variables and arrays are stored in memory based on the storage option you choose. This information can help you make better use of the space available in memory, and it can help you make speed/size and capacity tradeoffs in your BASIC code.

Table 15.1 Storage of Data in BASIC

Type of BASIC data	Compiler options	Memory location
Simple numeric[1] variables	All	DGROUP
Variable-length string descriptors	All	DGROUP
Variable-length string data	Far strings (/Fs)	Far heap
Variable-length string data	Default (not/Fs)	DGROUP
All numeric[1] array descriptors	All	DGROUP
Static numeric[1] array data	All	DGROUP
Dynamic numeric[1] array data	All	Far heap
Huge dynamic numeric[1] array data	Huge arrays (/Ah)	Far heap

1 Denotes **INTEGER, LONG, SINGLE, DOUBLE, CURRENCY**, and user-defined data types and fixed-length strings.

Note When you compile from within QBX and a Quick library is loaded, the default compile option is far string (/Fs). When you compile from the command line, near string storage is the default, as it was in all previous versions of BASIC. To use far string storage instead, add the /Fs option to the BASIC Compiler (BC) command line.

Since the QBX environment treats all strings as far strings, a program that uses near pointers to string data cannot run or be debugged in the QBX environment. However, the program may still work when compiled using the near string compiler option, and can be debugged with CodeView. On the other hand, far pointers will always work in QBX and in a program compiled with far strings. For detailed information about string storage, see Chapter 11, "Advanced String Storage."

String Array Storage

A string array has three parts in memory: the array descriptor, the array of string descriptors, and the string data. The array descriptor is always stored in DGROUP. Each element in the array of string descriptors is stored in DGROUP and contains the length and location in memory of the string data. The string data resides in the near heap if near string storage is specified at compile time, or in far heap if far string storage is specified at compile time. The 4-byte string descriptor for each variable-length string resides in DGROUP regardless of which string option (near or far) is used. Therefore, even with far strings, it is possible to run out of DGROUP by filling it with string descriptors. When this happens, the only remedy is to reduce the number of elements in the array.

BASIC string arrays can be either static or dynamic. A "static string array" is an array of variable length strings whose descriptors reside in a permanently allocated array in DGROUP. This array of descriptors is fixed when compiled and cannot be altered while a program is running.

A "dynamic string array" is a string array whose array of descriptors can be defined or changed during run-time with BASIC's **DIM**, **REDIM**, or **ERASE** statements. Dynamic arrays that are declared local to a procedure are de-allocated when control leaves the procedure. As with static string arrays, dynamic string array descriptors also reside in DGROUP, but they may change in number and/or location during execution of a BASIC program.

The location of the string data itself is independent of the static or dynamic status of a string array, and depends solely on whether the near or far string option is chosen when compiling.

Numeric Array Storage

Unlike string arrays, numeric arrays have a fixed amount of data associated with each element. All of the following information about numeric arrays is equally applicable to arrays of fixed-length strings or arrays of user-defined types, since they also have a fixed amount of data associated with each element in the array.

A numeric array has two parts: an array descriptor and the numeric array data itself. The array descriptor contains information about the array including its type, dimensions, and the location of its data in memory. Array descriptors are always stored in DGROUP. The data in a numeric array may be stored entirely in DGROUP or entirely in the far heap, as shown in Table 15.1.

As with string arrays, numeric arrays may be static or dynamic. A static numeric array has a fixed number of elements that are allocated when the program is compiled, while a dynamic numeric array is allocated and can be changed when the program is running. The compiler blocks out portions of DGROUP to contain all static array data, and this DGROUP space cannot be reclaimed while the program is running. Dynamic arrays are more flexible. They can be declared with a variable argument to the **DIM** statement, re-sized using the **REDIM** statement, and de-allocated altogether using the **ERASE** statement. Any space allocated for local dynamic arrays is reclaimed when BASIC leaves a particular procedure.

Certain tradeoffs are involved in the choice between static and dynamic numeric arrays. As described previously, static arrays are unchanging and thus consume a constant amount of memory. Dynamic arrays, however, will reclaim memory space after an array is erased or redimensioned to be smaller, thus allowing the same far heap space to be used for different purposes at different times in the same program. Static arrays provide the advantage of faster referencing and fixed locations in memory (helpful for mixed-language programming and quick look-ups).

Example In the following example, `A` and `B` start out the same size, but `A` can grow or shrink during program execution; if `A` is no longer needed, the space can be reclaimed:

```
Index = 0
DIM A (Index) AS INTEGER        ' A is dynamic,
DIM B (10) AS SINGLE            ' B is static,
Index = 20
REDIM A (Index) AS INTEGER      ' A's size can change,
.
.
.
ERASE A                         ' Or disappear.
```

Huge Dynamic Array Storage

Huge dynamic arrays allow you to create and store in memory arrays that contain more than 64K of data (64K is the limitation of standard dynamic arrays). Huge arrays are invoked by using the /Ah option when starting QBX or by using the /Ah option when compiling a program from the command line.

Huge arrays are limited to 128K unless the individual element size of each record divides evenly into 64K. In other words, the number of bytes in each element must be an integer power of two in order for an array of these elements to be larger than 128K, as shown by the following example:

```
TYPE ElementType
   A AS INTEGER          ' 2 bytes
   B AS LONG             ' 4 bytes
   C AS SINGLE           ' 4 bytes
   D AS DOUBLE           ' 8 bytes
   E AS CURRENCY         ' 8 bytes
   F AS STRING * 6       ' 6 bytes
END TYPE                 'Total of 32 bytes in ElementType
MaxElement% = 5000
DIM HugeArray(1 to MaxElement%) AS ElementType
```

The `ElementType` defined in the preceding example is a total of 32 bytes per element, so it will divide evenly into 64K and will enable BASIC to create the 160,000 byte array—assuming the /Ah option on QBX or BC was used, and there is sufficient far heap available). However, if the fixed-length string `F` is changed to either 5 or 7 bytes, a `Subscript out of range` error will result.

Note When compiling from within QBX, the /Ah compile option is automatically set only if you started QBX with the /Ah option. This can be changed only by exiting QBX and restarting it with the desired option.

Using the /Ah option in QBX or when compiling forces all dynamic arrays in your program to be huge. Huge arrays have slower access times and require more memory than similar standard dynamic arrays.

Tips on Conserving Data Space

The following are some tips on coding style that will help to conserve data space:

- Use constants.

  Normal numeric variables in BASIC programs reside in DGROUP. However, when a program is compiled, constant values can be folded directly into the generated code, so no additional DGROUP is needed to support these constants. If a value does not need to change in a program, constants can frequently make a program execute faster and will use less memory than variables.

- Use local variables in SUB procedures.

  Local variables within a SUB procedure occupy DGROUP space only while the program is running within that SUB procedure. BASIC keeps local variables on the stack, so that when the program returns from a procedure using local variables, the local stack variables are discarded and the stack is returned to the state it was in before the procedure was invoked. This is different from static variables, that occupy DGROUP space for the entire duration of the program.

- Trade file I/O buffer size for string space.

  File I/O buffers reside in the same area of memory as variable-length strings. If a program is compiled with /Fs, then the file I/O buffer will reside in far memory. Otherwise, the file I/O buffer will reside in DGROUP. The size of the file I/O buffer is defined by the **LEN=** argument of the **OPEN** statement, with the default buffer for sequential file I/O equal to 512 bytes (default for random file I/O is 128 bytes). A larger buffer generally requires fewer disk accesses and results in a faster executing program; but a smaller file I/O buffer takes up less memory in either DGROUP or the far heap, depending on the string option chosen when compiled.

Controlling Program Size

BASIC programs consist of the BASIC run-time routines and code generated by the compiler. Knowing how to manage both of these will help you to keep program size down leaving more room for other data in memory and on disk.

The BASIC run-time is the set of routines that support most of BASIC's underlying functionality. Since most compiled programs do not need all of the functionality available in the BASIC run-time routines, these routines are divided into many individual object modules and are included as necessary in the user's final program. The extent to which these routines are divided into individually accessible pieces is called "granularity."

The BASIC run-time routines included in any given program can be minimized either through writing code that does not use certain BASIC functionality, or through the use of "stub" files. This will result in smaller executables in memory and on disk, when compiling stand-alone programs.

Using Stub Files

Stub files are the special object files shipped with Microsoft BASIC that block certain pieces of the BASIC run-time from being included in the final executable file when the program is linked. To understand how stub files work, it is necessary to understand what happens during the link step of the build process.

When a BASIC source file is compiled into an object file, the object file contains many unresolved references. These references are simply calls into the BASIC run-time or other libraries on which the main BASIC program is depending, but are not present in the object file. During linking, the Microsoft Segmented-Executable Linker (LINK) matches up these calls to external procedures with the procedures themselves. LINK can be thought of as the matchmaker between the requests for certain functionality and the procedures that provide that functionality.

In order to resolve these calls to external procedures, LINK first searches every object module that is in the object field of the LINK command. Then, if there are any remaining unresolved references, LINK searches the files in the library field of the LINK command. The main difference between the object field and the library field, besides the order in which they are searched, is that every module found in the object field will be linked into the final executable file, while only those modules that contain procedures necessary to resolve references will be linked in from the library field. Therefore, if you specify a module in the object field that resolves the references that would otherwise be resolved by a module in the library field, you can block the module in the library field from being included in the final executable file.

Stub files accomplish that goal. Stub files contain dummy procedures, or "stubs," that have the same names as the BASIC run-time routines targeted for exclusion. The only functionality that exists in most of these dummy procedures is the ability to return a `Feature Stubbed Out` error message. Some stub files, however, contain a smaller, limited version of the functionality (as in SMALLERR and NOEDIT). Trading full-functionality routines for reduced or removed functionality routines reduces executable size.

Stub files can be used either when the stand-alone library or run-time module is created with BUILDRTM (see Chapter 21, "Building Custom Run-Time Modules") or the Setup program (see *Getting Started*), or they can be used individually with each program on a case-by-case basis. Stub files are only of value on a case-by-case basis when compiling stand-alone executables (compiling with the /O option).

It is important to remember to use the /NOE option when linking with stub files. This option causes LINK to ignore conflicts when a public symbol has been redefined, and instructs LINK to use only the specified object files (as opposed to using predefined dictionaries to identify libraries which will resolve references, as would happen if stub files were not being used).

The following example builds a program called MYPROG.BAS into an executable file and excludes support for COM and LPT I/O:

```
BC /O MYPROG.BAS;
LINK /NOE MYPROG.OBJ NOCOM.OBJ NOLPT.OBJ;
```

If you are certain all of your programs won't need some piece of functionality for which a stub file is provided, removing the functionality during setup is a way to reduce every program's size and thus increase room for other code or data competing for far memory space.

Note

Functionality that is excluded from the BASIC run-time modules during setup using stub files will not be available in any program, so these choices should be made cautiously. If you accidentally exclude needed functionality from the BASIC libraries during setup, run the Setup program again to add functionality back into the run-time module. See *Getting Started* for more information.

Table 15.2 lists the stub files that are shipped with Microsoft BASIC.

Table 15.2 BASIC Stub Files

File	Description
NOFLTIN.OBJ	Allows a program to contain **INPUT**, **VAL**, and **READ** statements without support for parsing floating point numbers that would require the floating point math package to be included. If a program is linked with this stub file, all numbers recognized by **INPUT**, **VAL**, and **READ** must be legal long integers.
NOEDIT.OBJ	Reduces functionality of the editor provided with the **INPUT** statement to support only Enter and Backspace keys (no Home, End, etc.).
NOCOM.OBJ	Removes support for COM device I/O. COM support is included by LINK if an **OPEN** statement is used with a string variable in place of the file or device name, as in `OPEN A$ FOR OUTPUT AS #1` or if a string constant starting with COM*n* is used with an **OPEN** statement.
NOLPT.OBJ	Removes support for LPT device I/O. LPT support is included by LINK if an **OPEN** statement is used with a string variable in place of the file or device name, as in `OPEN A$ FOR OUTPUT AS #1` or if a string constant starting with LPT*n*: is used with an **OPEN** statement. Using this stub file also removes run-time support for screen printing using Ctrl+PrtSc.
NOEVENT.OBJ	Removes support for event trapping. This stub file is only effective if linked with the run-time module; it has no effect when linked into stand-alone executables.
NOEMS.OBJ	Prevents a program linked for overlays from using Expanded Memory Specification (EMS); instead, the program will be forced to swap to disk.
OVLDOS21.OBJ	Required in order for a program linked for overlays to work under DOS 2.1. Does not reduce the size of the executable.
NOISAM.OBJ	Removes ISAM functionality from BASIC run-time modules. This stub file is not useful when creating stand-alone executable files.
SMALLERR.OBJ	Reduces length of run-time error messages.
87.LIB	Removes software coprocessor emulation, so that an 8087-family math coprocessor must be present in order for the program to perform any floating-point calculations, if floating-point statements are used in the program.

Table 15.2 *Continued*

File	Description
NOTRNEM*m*.LIB[1]	Removes support for any command using transcendental operation including: **LOG, SQR SIN, COS, TAN, ATN, EXD, ^, CIRCLE** statements with a start and /or stop angle, **DRAW** statements with **A** or **T** commands.
TSCNIO*sm*.OBJ[1,2]	These stub files limit the program to text-only screen I/O with no support for special treatment of control characters. Each TSCNIO*sm*.OBJ stub file is a superset of all the graphics-related stub files that follow.
NOGRAPH.OBJ	Removes all support for graphics statements and non-zero **SCREEN** modes. This stub file is a superset of all the following graphic **SCREEN** mode stub files.
NOCGA.OBJ	Removes all support for CGA graphics **SCREEN** modes 1 and 2.
NOHERC.OBJ	Removes all support for Hercules graphics **SCREEN** mode 3.
NOOGA.OBJ	Removes all support for Olivetti graphics **SCREEN** mode 4.
NOEGA.OBJ	Removes all support for EGA graphics **SCREEN** modes 7–10.
NOVGA.OBJ	Removes all support for VGA graphics **SCREEN** modes 11–13.

1 *m* can be either R or P for real or protected mode.
2 *s* can be either N or F for near or far strings.

Exploiting Run-Time Granularity with Code Style

Compiled BASIC programs will automatically leave out pieces of the run-time that LINK can determine are unnecessary. Since the run-time is pulled into a program in discrete pieces, understanding how to avoid pulling in large chunks of run-time support when they are not completely necessary can result in significantly smaller executable files, especially with smaller programs.

Avoid Accidental Use of Floating-Point Math

Depending on which math option is specified when compiling (/FPa for alternate math or /FPi for coprocessor/emulation), inadvertent inclusion of a floating-point math package can add over 10K to program size. Here are some tips to help you avoid pulling in a floating-point math package when it is not needed:

- Use integers.

  Start each program with `DEFINT A-Z`, or be sure to use integer variables (terminated with %) wherever possible. This will help you avoid the floating-point math package and speed up your program. Since the default data type in BASIC is SINGLE (4-byte floating-point real), people frequently make the mistake of using the SINGLE data type where integers would actually better suit their needs.

- Use the integer division operator (\) whenever doing integer division.

  This division operator will not cause the floating-point math support to be pulled in if the math pack is not needed elsewhere in the program. Using the regular division operator (/) always pulls in the floating-point package, even when used with integers.

- Avoid BASIC statements and functions that use the floating-point package.

  Some of the BASIC statements and functions that pull in the floating point package are obvious, such as SIN, COS, TAN, ATN, LOG, and EXP. The not-so-obvious BASIC functions that use the floating-point package are VAL, WINDOW, DRAW, TIMER, RANDOM, INPUT, READ, PMAP, POINT, PRINT USING, and SOUND.

Use Constants in SCREEN Statements

Using a variable as an argument to the SCREEN statement will cause support for all graphic screen modes to be pulled in, unless stub files are being used, whereas a SCREEN statement with a constant only pulls in the support necessary for the specified screen mode.

Minimizing Generated Code

As a BASIC program gets longer, the compiled executable file becomes increasingly dominated by generated code. Since run-time library routines are included only once in any given program, the larger a program becomes, the more it benefits from the BASIC strategy of using run-time library routines for much of its functionality. The converse is also true—the shorter a program is, the more its size is dominated by the run-time routines, making generated code size less important. However, if a program is compiled without the /O option, no run-time library routines are contained in the executable file, so the marginal space taken up on disk is dominated by code generation even for short BASIC programs.

The following are some tips to help you keep your programs small by minimizing generated code:

- Use procedures.

  Designing a BASIC program to efficiently re-use code for repetitive operations will not only make your code smaller but it will also make it more understandable, easier to maintain and debug, and easier to use in other programs.

- Beware of event trapping.

  While it takes relatively little BASIC code to set up event trapping, the resultant compiler-generated code will be disproportionately large, since the compiler must generate tests for the specified event between each label or statement. Event-trapping code can be more precisely controlled with **EVENT ON** and **EVENT OFF** statements, which allow you to specify exactly where within a module event-trapping code is generated.

- Compile modules that use **ON ERROR RESUME** statements separately.

  BASIC does not provide statements to turn error trapping on and off. Compiling with the /X option (to support **ON ERROR RESUME** statements) generates 4 bytes of code per BASIC statement in a module compiled with /X. These 4 bytes contain the location to which the program should return if an error occurs at any particular time. To get the smallest size and best performance in a program with such error trapping, use local error handling and have all those procedures with local error handling in a separate module compiled with /X. This will prevent the performance degradations associated with /X from affecting those pieces of your program that do not require error handling.

Overlays

Modular programs that use overlays can execute in substantially less memory than programs that do not use overlays. The key to successful program design for overlays is minimizing the performance penalty inherent in swapping modules in and out of memory. This can be accomplished by designing the program in groups of interconnected modules that will primarily call each other, with only infrequent calls outside the group that would cause another overlay to be swapped back into memory.

In general, event-handling routines and general-purpose routines should go in code which is not overlaid, unless events are expected to be rare. See Chapter 18, "Using LINK and LIB," for more information about using overlays in BASIC.

Minimizing Executable File Size

Compiling without /O is the easiest single thing which can be done to minimize the size of BASIC executables on disk. The resultant executable programs do not contain the BASIC run-time routines. Rather, these programs access the same run-time routines that exist in a single BASIC run-time file on disk (one of the files named BRT70*mso*, where *mso* stand for the math, string, and operating mode options). You can customize these run-time modules by adding modules of your own routines with the BUILDRTM utility. See Chapter 21, "Building Custom Run-Time Modules," for more information on how to use this feature.

The primary tradeoff of compiling programs that require the presence of a BRT70*mso*file is that the program will not work as a stand-alone, single-file executable. The ultimate end-user of the program must always have a copy of the appropriate BRT70*mso* file present in order for the program to run. Other tradeoffs made in working with run-time modules are that these programs use more memory (since the entire run-time gets loaded into memory) and program initialization is slower.

If the program is meant to be stand-alone and is compiled with /O, then all of the BASIC run-time management tips in the section "Controlling Program Size" earlier in this chapter are equally valuable in reducing the size of executables on disk. Minimizing generated code will also result in proportionately smaller executable files.

LINK provides some options such as /EXEPACK, /PACKCODE, and /FARCALLTRANSLATION that help to compress the size of the executable and improve executable speed in some cases. See Chapter 18, "Using LINK and LIB," for more information on these and other LINK options.

Compiling Programs for Speed

This section and the rest of the sections in this chapter discuss ways to improve the performance of your BASIC programs so they execute in the shortest amount of time possible.

Compiling for the Target System

Knowing the hardware configuration of the system on which the program will ultimately run makes it possible to compile a program optimally for speed on that particular machine.

The /G2 compiler option causes the compiler to generate code that takes advantage of the expanded instruction set of the Intel 80286 microprocessor. If the target system is definitely an IBM AT or compatible 286-based machine, the /G2 option will make compiled BASIC programs smaller and faster. PCs with backward-compatible microprocessors that support the 286 instruction set (i.e. 386 and 486) will also run programs compiled with /G2 faster than programs compiled without /G2.

Math Options

Depending on whether or not the target system has an 8087-family math coprocessor, one of two math packages will provide the fastest possible floating-point math support.

80x87 Support

If the target system is expected to have an 80x87-family math coprocessor, then compiling with the /FPi option (the default) will either precisely emulate the functionality of such a chip or will use the hardware directly to perform the desired functions. If you are absolutely certain that the target system will have such a coprocessor, then you may wish to LINK with 87.LIB (the coprocessor math package which does not provide floating-point operation software). If math is used, this reduces executable size by over 9 K.

Alternate Math

If the target system is not expected to have an 80x87-family math coprocessor, then compiling with the /FPa option will link in the alternate floating-point math pack, which does not attempt to emulate the 80x87 hardware. Alternate math is an IEEE-compatible math package optimized for speed and size on systems without a floating-point coprocessor. The tradeoff of using the alternate math package is that a small amount of precision is lost compared with the true 80x87 emulator. For most applications, this loss of precision is negligible and the alternate math package will offer better math speed with no undesirable side effects.

Specifically, all floating-point calculations performed by alternate math are performed in either 24-bit or 53-bit mantissa precision for **SINGLE** or **DOUBLE** data types, respectively. With the 80x87/emulator, all intermediate results in an expression are calculated to 64-bits of mantissa precision.

Simple operations involving only one calculation followed by an assignment, such as the following will yield the same result with either math package:

```
A! = B! + C!
A# = B# + C#
```

The differences start appearing when multiple calculations are performed in a single expression, as in the following example:

```
B! = 1000001!
C! = 1!
D! = 1!
A! = (1000001! * B! + C!) * D! - (1000001! * B!)
```

With the alternate math package, the intermediate results are calculated to 24 bits of precision. The result of this expression to 24 bits is 0. With the 80x87/emulator, the result is exactly 1. The preceding single-precision calculation could be performed without losing any bits of significance.

Transcendental functions computed with the emulator are accurate to between 62 and 64 bits. The alternate math package is also accurate to within 2 bits of its normal mantissa precision for each data type in transcendental math. This means that the following expression could be accurate to 22 bits with alternate math, but with the 80x87/emulator it will usually be accurate to 24 bits (it would be off by 1 bit in about 1 in 2^{38} samples):

```
A! = SIN(1.0!)
```

Managing Data for Speed

Variable types and storage options have a significant impact on program execution speed. Choosing data types and storage options intelligently can decrease execution time.

Constants

Use constants whenever possible to save space and make programs faster. References to named constants are resolved when compiled and the values are then embedded directly into the generated code (except for string and floating point constants). Where constants can be used, they are almost universally more efficient than variables.

Integers

Integers are the fastest data type available in BASIC. Integers are smaller than any other numeric BASIC data type (each integer uses only 2 bytes per instance), and integers can be handled efficiently without run-time calls through compiler-generated in-line code. A good programming habit is to put `DEFINT A-Z` at the top of each module and procedure and then only use other data types as necessary.

The following tips on using integers can help your programs run faster:

- Loop variables should usually be integers.

  A common error made by BASIC programmers is to just use the default data type (**SINGLE** 4-byte floating point real) for loop variables. This error could slow down loop speed significantly and cause program size to grow by unnecessarily pulling in a floating-point math package if it hasn't yet been pulled in.

- Booleans should always be integers.

  If a variable's purpose is to either specify true or false, or some other discrete set of values, integers will be by far the smallest and fastest data type suitable for this purpose.

- BASIC statement and function arguments should be integers wherever possible.

  Using integer arguments for calls to graphics functions, for example, will make the code execute more quickly and will not require the use of the floating-point math package.

Currency

Microsoft BASIC supports a new data type, **CURRENCY** (specified with the **@** suffix), that can be thought of as an extra-long integer that has been shifted to accommodate fixed decimal-place fractional values. Internally, the **CURRENCY** data type is represented as an 8-byte integer that is then scaled by a factor of 10,000 to leave four decimal places to the right of the decimal point, and 15 places to the left. Its internal representation as an integer gives the **CURRENCY** data type a significant advantage in speed over floating point for addition and subtraction and a disadvantage in speed for other mathematical operations. Where fixed-decimal precision across the **CURRENCY** data type's range of values is acceptable (for example in dollar and cent calculations), **CURRENCY** should be seen as a desirable alternative to traditional floating point data types.

Compared with binary-coded-decimal (BCD) data types, the **CURRENCY** data type has superior range for a comparable number of bytes (since BCD data type only uses 100 out of 256 values in every byte, while **CURRENCY** uses all 256). The **CURRENCY** data type is also faster in math operations such as addition and subtraction.

Near Vs. Far Strings

While far strings provide greater capacity, near strings are measurably faster for most operations, since it takes less time to access a near string than a far string. Depending on the program, string speed or capacity may be most desirable, and the choice between string packages should be made accordingly.

Static Vs. Dynamic Arrays

As with type of strings, the type of arrays used involves a tradeoff of capacity for speed. Static array referencing is always faster than referencing dynamic arrays, but dynamic arrays (since they exist in far memory) offer significantly higher capacity. Similarly, accessing a standard dynamic array is faster than accessing a huge array (compiled with the /Ah operator).

Optimizing Control-Flow Structures for Speed

BASIC provides three different levels of general purpose control-flow structures, each with its own advantages. They are the **GOTO**, **GOSUB/DEF FN** procedures, and **SUB/FUNCTION** procedures. The speed with which each of these can cause your program to branch off to execute different pieces of code is largely a function of how much information must be passed and how much of the current context must be maintained.

GOTO

The **GOTO** statement is the oldest and least structured of the BASIC branching controls, but is still unmatched in raw jumping speed. The simple mapping of the **GOTO** statement onto the single-instruction **JMP** in assembly language makes the **GOTO** statement the fastest way to go from point A to point B in a program. The obvious tradeoff here is that **GOTO** statements are unstructured, making code difficult to read, debug, and maintain, and are therefore considered poor programming style.

GOSUB and DEF FN Subroutines

GOSUB and **DEF FN** procedures do little more for the user than the **GOTO** statement, except to push a return address onto the stack so that after the procedure is finished, a return instruction is executed that returns the program to the statement after the assembly language **CALL** instruction. This branching construct is slower than the **GOTO** statement, but it adds the advantage of returning to the point after the branch.

SUB and FUNCTION Procedure Calls

SUB and **FUNCTION** procedures are the powerful, fully structured BASIC constructs that allow passing of parameters, local variables, recursion, local error handling, inter-module calls, and mixed-language programming. The elegance of these modular language features leads some people to argue that the older constructs discussed above are completely obsolete, despite their speed advantage in some situations. The speed of a call to a procedure of this type is dominated by the amount of context and parameter information that must be stored on the stack (called the stack frame) before the branch can take place. The size of this stack frame is determined by the number and size of parameters being passed to the procedure, the amount of error-trapping information to be maintained, and the code that exists within the procedure itself.

The following tips help you improve the speed of calls:

- Minimize parameter passing.

  While parameter passing is generally preferred programming style compared to using global variables, pushing parameters on the stack takes time. When call speed is critical, global variables reduce frame size and minimize the net time spent transferring control to a procedure.

- Compile with the /Ot option when appropriate.

  Compiling with the /Ot option will improve call speed if you are not also compiling with the /D or /Fs option and your program does not use local error handling. If a **SUB** or **FUNCTION** procedure uses local error handling, calls a **DEF FN** or **GOSUB** procedure, or contains a **RETURN** statement, then a full stack frame is generated and compiling with the /Ot option will not improve call speed.

- Avoid setting up a large stack frame.

  A BASIC stack frame that contains a large block of information about the current context of the program will be saved in some situations as soon as a **SUB** or **FUNCTION** procedure is invoked. The following things require a large stack frame and should be avoided if call speed is critical:

  - **GOSUB** and **DEF FN** invocations from within procedures
  - Module-level error handlers (if only procedure-level error-handling routines are used, a BASIC stack frame is required only when calling those procedure containing local error-handling routines)
  - Run-time error checking (compiling with the /D option for debug information)
  - **ON...GOSUB** statements within procedures
  - Compiling with the /Fs option

- Pass parameters by value.

  When passing dynamic array elements or fixed-length strings as parameters to BASIC procedures, putting parentheses around the expression within the parameter list will cause the parameter to be passed by value rather than by reference, which will result in faster calls. Also, when calling a non-BASIC procedure from BASIC, using the **BYVAL** attribute in the parameter list of the **DECLARE** statement will cause the particular parameter to be passed by value, thus speeding up the call. Of course, passing parameters by value should be used only when the value of the parameter does not need to be changed globally within the procedure.

- Use **STATIC SUB** statements to maximize call speed to **SUB** procedures.

  Variables will then be static by default and will exist between calls to the **SUB** procedure; the compiler won't have to recreate them. The variables will require more space, however.

Writing Faster Loops

While looping is a straightforward task in BASIC, there are some things that can help speed up simple looping. The first thing to remember, once again, is always use integer loop counters. Second, remember to remove any and all "loop invariant code"— code that doesn't execute differently during each iteration of the loop. Any operation whose result does not change through multiple iterations of the loop should be coded outside of the loop altogether. An example of removal of loop invariant code is:

```
FOR I%=1 to LastLoop%
   A(I%)=SIN(B)
NEXT I%
```

The preceding example could be more efficiently recoded as:

```
C=SIN(B)
FOR I%=1 to LastLoop%
   A(I%)=C
NEXT I%
```

Optimizing Input and Output for Speed

File and screen I/O are frequently the speed bottlenecks for BASIC programs, because of the relatively slow speed of disk access and writing to screen. But as with all the other areas of BASIC programming, there are ways to maximize the performance of these operations.

Sequential File I/O

The **OPEN** statement has an optional **LEN=** argument that can be used to specify the buffer size that will be used when opening a file for sequential file I/O. This sequential file I/O buffer resides in the same area of memory that contains variable-length strings, namely DGROUP if the program is compiled for near strings, or the far memory if the program is compiled with /Fs. The larger this buffer is, the smaller the number of disk accesses that will be necessary to perform a given quantity of file I/O with the disk. The tradeoff here is speed of file I/O in exchange for capacity in the area of memory (near or far) that contains variable length strings. The default sequential buffer size is 512 bytes.

ISAM

The BASIC ISAM statements and functions make structured file I/O much faster and simpler than using non-ISAM file I/O statements and functions and creating your own code in BASIC to handle structured file access. ISAM is optimized for working with very large numbers of records that must be accessed by indexes that do not necessarily correspond to the physical order of the records in the file. When using ISAM, however, compiling with /D can degrade performance. See Chapter 10, "Database Programming with ISAM," for more information on programming with the ISAM statements and functions.

Printing to Screen

The stub files TSCNIO*sm*.OBJ not only reduce executable size but also speed up printing characters to the screen. These stub files eliminate checking for special control characters when they are printed, so that all characters are then sent directly to the screen without filtering out and treating characters below ASCII 32 differently from the other characters.

Sending text to the screen can be accelerated further through precise use of the **PRINT** statement. The **PRINT** statement comes with many automatic functions, among them are auto-scroll if there is no terminating semicolon. Putting a semicolon at the end of any **PRINT** statement where a line feed is not needed will save an unnecessary call to the lower level print routines to start a new line. This code change could make printing to screen twice as fast.

Other Hints for Speed

The following sections include some general hints to help you improve program execution speed.

Event Trapping

Event trapping in BASIC programs forces the compiler to generate event-checking code that is executed either between statements or between labels. This makes the code larger and slower. Precise use of **EVENT ON** and **EVENT OFF** statements to limit the areas affected by event trapping, or avoiding use of event trapping altogether when possible, will substantially improve speed of the resultant executable.

Line Labels

Excessive numbers of line labels limit the optimizations that can be performed by the compiler. Programs run faster after extraneous line labels have been removed. The REMLINE.BAS program that comes with Microsoft BASIC can automatically remove all unneeded line labels and thus help the compiler perform more sophisticated optimizations, resulting in faster executables.

Buying Speed with Memory

A common technique used to gain speed in calculation-intensive programs is to set up arrays in which the results of a particular function or calculation, across the expected range of input values, are stored. The underlying assumption is that the time needed to refer to the array is shorter than the time needed to perform the calculation itself, and that the speed advantage is worth the price of additional memory used to store the array.

For example, a program that repeatedly requires the tangents of angles to be calculated might run substantially faster if builds an array of values `TAN(x)` that can be quickly referred to, rather than calculating the tangent each time it is required.

Part 2

Using BASIC Utilities

Part 2 describes how to use the compiler and utilities provided with Microsoft BASIC.

Chapter 16 describes the features, options, and usage of the BASIC Compiler (BC).

Chapters 17–19 discuss topics related to linking programs and creating Quick and object-module libraries. Chapter 17 provides an overview of linking and libraries for BASIC and mixed-language programs. Chapter 18 describes the usage and options for the Microsoft Segmented-Executable Linker (LINK) and the Microsoft Library Manager (LIB). Chapter 19 gives instructions for creating a Quick library and describes how to load and use these libraries within the QBX environment.

Chapter 20 contains a discussion of NMAKE, a program maintenance utility.

Chapter 21 describes the BASIC Run-Time Module Builder (BUILDRTM) and how it can be used to create custom run-time modules.

Chapter 22 describes the structure of an online help file, and how you can create or modify these files using the HELPMAKE utility.

Chapter 16
Compiling with BC

This chapter explains how to compile your BASIC program using the BASIC Compiler (BC). You'll learn how to invoke BC from the command line as well as how to use command-line options to BC.

While you can compile and link your program from QBX by choosing Make EXE File from the Run menu, you might run BC from the command line to create an object file. Then you can run the LINK utility to link object modules and libraries to create an executable file. You may want to invoke BC and LINK in separate steps for the following reasons:

- To compile a program that is too large to compile in memory within the QBX environment
- To debug your program with the Microsoft CodeView debugger
- To use a different text editor
- To create listing files for use in debugging a stand-alone executable program
- To use options not available within the QBX environment, such as storing arrays in row order (/R)
- To link your program with stub files, which reduce the size of executable files in programs that do not use a particular BASIC feature

This chapter assumes that you will be using BC from the command line. If you are compiling your program from QBX, use online Help to learn how to invoke the compiler and specify compiler options.

New Options and Features

The following options and features have been added to this release of BC:

- The /Fs option enables you to store string data in far memory.
- The /G2 option generates instructions specific to the 80286 memory chip that result in smaller, faster executable code.
- The /Ot option optimizes the performance of procedure calls.
- The /I*x*: options control memory use in ISAM applications.
- The DOS INCLUDE environment variable enables you to determine where BC will look for included files without changing the **$INCLUDE** metacommand in your source file.

Compiling with the BC Command

You can compile with the BC command in either of the following ways:

- Type all information on a single command line, using the following syntax:

  BC *sourcefile* [[,[[*objectfile*]][,[[*listingfile*]]]] [[*options*]][[;]]

  You can let BC create default object and listing files for you by typing a semicolon or commas after *sourcefile*. BC will generate an object file with the same name as your source file (without its extension), and will append an .OBJ (object file) or .LST (listing file) filename extension.

- Type the BC command and respond to the following prompts:

```
BC
Source Filename [.BAS]:
Object Filename [filename.OBJ]:
Source Listing [NUL.LST]:
```

  The argument *filename* is the name of your source file without the .BAS filename extension.

  To accept the default filenames shown in brackets, type a carriage return or semicolon after the colon.

Table 16.1 shows the input you must give on the BC command line or in response to each prompt.

Table 16.1 Input to the BC Command

Field	Prompt	Input
sourcefile	Source Filename	The name of your source file. If USER is specified, input is taken from the keyboard.
objectfile	Object Filename	The name of the object file you are creating.

Table 16.1 *Continued*

Field	Prompt	Input
listingfile	Source Listing	The name of the file containing a source listing. The source-listing file contains the address of each line in your source file, the text of the source file, its size, and any error messages produced during compilation. If USER is specified, the listing is sent to the screen.
options	None. You may enter options after any response.	Any of the compiler options described in "Using BC Command Options" later in this chapter.

Specifying Filenames

The BC command makes certain assumptions about the files you specify, based on the paths and extensions you use for the files. The following sections describe these assumptions and other rules for specifying filenames to the BC command.

Uppercase and Lowercase Letters

You can use any combination of uppercase and lowercase letters for filenames; the compiler accepts uppercase and lowercase letters interchangeably. Thus, the BC command considers the following three filenames to be equivalent:

```
abcde.BAS
ABCDE.BAS
aBcDe.Bas
```

Filename Extensions

The name of a DOS file has two parts: the base filename, which includes everything up to (but not including) the period (.), and the filename extension, which includes the period and up to three characters following the period. In general, the extension identifies the type of file (for example, whether the file is a BASIC source file, an object file, an executable file, or an object-module library).

BC uses the filename extensions described in the following list:

Extension	File description
.BAS	BASIC source file.
.OBJ	Object file.
.LST	Listing file produced by BC. The extension .LST is the default filename extension; it will be overridden if you provide an extension for *sourcefile*.

Paths

Any filename can include a full or partial path. A full path starts with the drive name; a partial path has one or more directory names preceding the filename, but does not include a drive name.

Giving a path allows you to specify files in different paths as input to the BC command and lets you create files on different drives or in different directories on the current drive.

Note For files that you are creating with BC, you can give a path ending in a backslash. When it creates the file, BC uses the default name for the file.

You can use the DOS command SET to change the INCLUDE environment variable determining a search path for files included in your BASIC source file. To do this enter a command with the following syntax before invoking BC:

SET INCLUDE = *path*;*path*...

Using BC Command Options

Options to the BC command consist of either a slash (/) or a dash (-) followed by one or more letters. (The slash and the dash can be used interchangeably. In this manual, forward slashes are used for options.) From QBX, you can modify BC options by choosing options in the Modify EXE File dialog box.

The BC command-line options are explained in the following table:

Option	Description
/A	Creates a listing of the disassembled object code for each source line and shows the assembly language code generated by the compiler.
/Ah	Allows dynamic arrays of records, fixed-length strings, and numeric data to occupy all of available memory. If this option is not specified, the maximum size is 64K per array. Note that this option has no effect on the way data items are passed to procedures.

/C:*buffersize*	Sets the size of the buffer receiving remote data for each communications port when using an asynchronous communications adapter. (The transmission buffer is allocated 128 bytes for each communications port and cannot be changed on the BC command line.) This option has no effect if the asynchronous communications card is not present. The default buffer size is 512 bytes total for both ports; the maximum size is 32,767 bytes.
/D	Generates debugging code for run-time error checking; enables Ctrl+Break; and adds implicit checkpoints after every **DELETE**, **INSERT**, **UPDATE**, or **CLOSE** ISAM statement. This option is the same as the Produce Debug Code option from the Run menu's Make EXE File command within the QBX environment.
/E	Indicates presence of **ON ERROR** with **RESUME** *linenumber* statements. (See also the discussion of the /X option in this list.)
/FPa	Causes your program to use the alternate math library for floating-point operations. See "Using Floating-Point Options" later in this chapter for more information about this option.
/FPi	Causes the compiler to generate "in-line instructions" for use in floating-point operations. See "Using Floating-Point Options" later in this chapter for more information about this option.
/Fs	Enables far strings in user programs. See the section "Using Far Strings" later in this chapter for additional information about this option.
/G2	Enables specific to the 80286 memory chip instructions. For computers containing an 80286 processor, using this option results in faster and smaller executable programs.
/I*x*:*number*	Controls memory management for ISAM. See Chapter 10, "Database Programming with ISAM," for detailed information about how to use this option.
/Lp	Creates a protected-mode object file; the default if running BC from an OS/2 protected-mode session.

/Lr	Creates a real-mode object file; the default if running BC from DOS or a real-mode OS/2 session.
/MBF	The intrinsic functions **MKS$**, **MKD$**, **CVS**, and **CVD** are converted to **MKSMBF$**, **MKDMBF$**, **CVSMBF**, and **CVDMBF**, respectively. This allows your BASIC program to read and write floating-point values stored in Microsoft Binary format.
/O	Substitutes the default stand-alone library for the default run-time library. For example, if the default run-time library is BRT70ENR.LIB, the /O option would substitute the stand-alone library BCL70ENR.LIB for it.
/Ot	Optimizes execution speed for **SUB**, **FUNCTION**, and **DEF FN** procedures.
/R	Stores arrays in row-major order. BASIC normally stores arrays in column-major order. This option is useful if you are using external routines written in another language that store arrays in row order.
/S	Writes quoted strings to the object file instead of the symbol table. Use this option when an `Out of memory` error message occurs in a program that has many string constants.
/T	Suppresses warnings given by the compiler. By default, the interpreter passes this option to BC when you select Make EXE File from the environment.
/V	Enables event trapping for communications (**COM**), lightpen (**PEN**), joystick (**STRIG**), timer (**TIMER**), music buffer (**PLAY**), function keys (**KEY**), and **UEVENT**. Use this option to check between statements for an occurrence of an event.
/W	Enables event trapping for the same statements as /V, but checks at each line number or label for occurrence of an event.
/X	Indicates presence of **ON ERROR** with **RESUME**, **RESUME NEXT**, or **RESUME 0**.
/Z	Produces a listing of compile-time errors in a form readable by the M editor. If used when compiling within M, /Z allows you to locate and scroll through errors in your source and include files by invoking the Nxtmsg editor function.
/Zd	Produces an object file containing line-number records corresponding to the line numbers of the source file. This option is useful when you want to perform source-level debugging using the Microsoft Symbolic Debug Utility (SYMDEB), available with the MASM version 4.0.
/Zi	Produces an object file of debugging information that can be used by the Microsoft CodeView debugger.

Note If you are compiling your program from QBX, not all BC options can be specified You can set the /Fs, /Lr, /Lp, /FPi, /FPa, /G2, /D, /O, /Ot, /S, and /Zi; options from the Make EXE File dialog box.

Using Far Strings (/Fs)

When you use the /Fs option to compile a module, be aware that the object file you create is not compatible with modules that were compiled without the /Fs option. For detailed information about how BC processes far strings in a module, see Chapter 4, "String Processing."

Using Floating-Point Options (/FPa and /FPi)

Microsoft BASIC offers two main methods for handling floating-point math operations: in-line instructions (/FPi) or alternate math library (/FPa). Your choice of methods affects the speed and accuracy of floating-point operations, as well as the size of the executable file.

Object files created with the /FPa option are not compatible with those created with the /FPi option. If you link files that use incompatible floating-point methods, BASIC detects the incompatibility and terminates with the error message `Error during run-time initialization`. When you link multiple-source files into a single executable file, you must ensure that every part of your program handles floating-point operations consistently.

In-Line Instructions (/FPi)

Specifying the /FPi option causes the compiler to create "in-line instructions" for use in floating-point operations. In-line instructions are machine-code instructions that a math coprocessor can execute. At run time, BASIC checks to see whether a math coprocessor is present. BASIC uses the coprocessor if it is present, or emulates its functions in software if it is not.

This method of handling floating-point operations provides the fastest solution if you have a math coprocessor, and it offers the convenience of automatically checking for a coprocessor at run time. For these reasons, BASIC uses in-line instructions (/FPi) by default if you compile without choosing a floating-point option.

If you choose the /FPi option, your program will link the emulator library (EMR.LIB or EMP.LIB), which provides a large subset of the functions of a math coprocessor in software. The emulator can perform basic operations to the same degree of accuracy as a math coprocessor. However, the emulator routines used for transcendental math functions differ slightly from the corresponding math coprocessor functions, causing a slight difference (usually within 2 bits) in the results of these operations when performed with the emulator instead of a math coprocessor.

Alternate Math Library (/FPa)

If you compile with the /FPa option, your program uses the alternate math library for floating-point operations. The alternate math library (BLIBFA.LIB or BLIBFP.LIB) is a software-only math package that uses a subset of the Institute of Electrical and Electronics Engineers, Inc. (IEEE) format numbers.

This option offers the fastest math solution if your computer does not have a math coprocessor. It also creates a smaller executable file. However, using the alternate math library does sacrifice some accuracy in favor of speed and simplicity because infinity, "not-a-number" (NAN) values, and denormal numbers are not used. See Chapter 15, "Optimizing Program Size and Speed," for information about the performance gain and accuracy of the alternate math library. See Appendix B, "Data Types, Constants, Variables, and Arrays," for the range of values valid with alternate math.

When you choose the alternate-math library, BASIC does not use a math coprocessor, even if one is present.

Run-Time Modules and Floating-Point Methods

Your choice of floating-point method has implications at run time for programs that use run-time modules and libraries. If you compile without the /O option, you must ensure that the appropriate run-time module is present at run time. (The /O option creates a stand-alone file that does not require the BASIC run-time module.) For each operating mode (real or protected), Microsoft BASIC offers several versions of the run-time module. Some versions use the in-line-instructions method for floating-point operations, while other versions use the alternate-math-library method. See Chapter 21, "Custom Run-Time Modules," for more information about run-time modules and libraries.

Optimizing Procedure Calls (/Ot)

Calls to **SUB**, **FUCTION**, and **DEF FN** procedures can be optimized by compiling with the /Ot option under certain conditions. The /Ot option reduces the size of the stack frame generated by certain calls, thereby reducing the amount of information passed to the stack when a call is made. Reducing the amount of information passed to the stack reduces the amount of time required to execute. The /Ot option will improve execution speed as follows:

- **SUB** and **FUNCTION** procedures

  A reduced stack frame is generated with /Ot if no module-level error-handling routines exist in the code and the /D or /Fs option is not used. The full stack frame is generated and no performance benefit results if your code uses local error handling, uses a **DEF FN** or **GOSUB** statement, has a return, or contains an **ON** *event* **GOSUB** statement.

- **DEF FN** procedures

  A full stack frame is generated and no benefit results if the /D, /Fs, /E, or /X option is used. A partial stack frame is generated if the /W or /V option is used. In all other cases, no stack frame is generated and a performance benefit results.

Chapter 17
About Linking and Libraries

This chapter provides an overview of library and linking support provided with Microsoft BASIC. You'll learn about the types of libraries you can work with and how to link them into your BASIC programs. If you are interested in learning about how to use the LINK and LIB utilities and their options, see Chapter 18, "Using LINK and LIB." The Quick library, a special type of library used within the QBX environment, is described in Chapter 19, "Creating and Using Quick Libraries."

What Is a Library?

A library is an organized collection of object code; that is, a library contains functions and data that are already compiled (with a compiler or an assembler) and are ready to link with your program. The structure of a library supports the mass storage of common procedures—procedures that can be called by a variety of programs. These common procedures, called "modules," can be added, deleted, changed, or copied.

Libraries are typically used for one of three purposes:

- To support high-level languages. For example, Microsoft BASIC performs input/output and floating-point operations by calling standard support routines. Because the support routines are available in a library, the compiler never needs to regenerate code for these routines.
- To perform complex and specialized activities, such as financial functions or matrix math operations. Libraries containing such routines often are provided by the makers of the compilers or by third-party software vendors. Microsoft BASIC comes with several such libraries.
- To support your own work. If you have created routines that you find useful for a variety of programs, you may want to put these routines into a library. That way, these routines do not need to be rewritten or recompiled. You save development time by using work you have already done, and disk space because you don't have to replicate source code.

Note Because LIB and LINK assign special meaning to the at sign (@), it should never be used as the first character of a filename that is to be made into an executable or library file.

Types of Libraries (.LIB and .QLB)

Microsoft BASIC provides tools for creating two different types of libraries, which are identified by the following filename extensions:

Extension	Description
.LIB	Characterizes an object-module library, one that is created with the Microsoft Library Manager (LIB). Object-module libraries can be linked with object modules to create an executable program or a Quick library.
.QLB	Characterizes a Quick library, a special kind of library that can be loaded and used in the QBX environment. Quick libraries are created by the Microsoft Segmented-Executable Linker (LINK).

You can think of a Quick library as a group of procedures appended to QBX when the library is loaded into the environment, while an object-module library is a collection of independent, compiled procedures. Object-module libraries can be linked with a main module to create a file that is executable from the DOS command line.

Both types of libraries are discussed in the following sections.

Object-Module Libraries (.LIB)

Object-module libraries can be linked with compiled object files to produce stand-alone executable programs. These libraries normally have the .LIB filename extension. Microsoft BASIC supplies the following types of object-module libraries:

- Stand-alone libraries. These libraries allow your executable file to run alone, without run-time modules (see the following item).Your program will look for this type of library during linking if you compiled your program with the /O option of the BASIC Compiler (BC).

- Run-time libraries. These libraries are used in conjunction with special modules called run-time modules. Your program will look for this type of library during linking if you have not used the /O option of BC when you compiled your program. Run-time modules and libraries are discussed in greater detail in Chapter 21, "Building Custom Run-Time Modules."

- Add-on libraries and BASIC toolbox files. Microsoft BASIC supplies two add-on libraries: a financial function library and a date/time library. These are object-module libraries (.LIB) that contain special-purpose routines.

  BASIC toolbox files are BASIC source files (.BAS) containing mouse, menu, window, graphics, and support routines. These files can be turned into stand-alone or Quick libraries. (See Chapters 18, "Using LINK and LIB" and 19, "Creating and Using Quick Libraries," for details about creating libraries.)

The Setup program automatically creates stand-alone and run-time libraries during the setup operation. The exact libraries created depend upon which math package, mode (real or protected), and string support options you specify.

You can also create your own custom library or turn BASIC source code modules into a library. To do this, separately compile source code modules that you want in the library, then use LIB to combine those object modules into one library. LIB also lets you add, delete, replace, copy, and move modules in the library as you choose. The QBX environment automates this process if you select the Make Library command from the Run menu.

Stand-Alone Libraries

Stand-alone libraries use the following naming convention:

BCL70 *float string mode*.LIB

The possible values for each variable are shown in Table 17.1.

Table 17.1 Library Naming Conventions

Variable	Letter	Feature
float	E	Emulator (/FPi) (default)
	A	Alternate math (/FPa)
string	N	Near strings (default)
	F	Far strings (/Fs)
mode	R	Real mode(/LR) (default)
	P	Protected mode (/LP)

For example, if you specify the emulator floating-point math, far strings, and real mode options during setup, your program would search for and link with the BCL70EFR.LIB stand-alone library. From within QBX, you can change the library used for linking your program by changing options in the Make EXE dialog box.

When you compile with the /O option of BC, an object file requiring a stand-alone library is produced. If you then run LINK, the proper stand-alone library is linked with your program to produce a stand-alone executable file. In the QBX environment, your program is automatically linked with a stand-alone library if you select the Stand-Alone EXE option from the Make EXE File dialog box.

Stand-alone programs require more disk space than those requiring the run-time module, and variables listed in **COMMON** statements as well as open files are not preserved when a **CHAIN** statement transfers control to another program. Stand-alone programs do have the following advantages, however:

- Stand-alone programs always require less memory than their run-time equivalents.
- Execution of the program does not require that the run-time module be on the disk when the program is run. This is important when you write programs that users will copy, since an inexperienced user may not know to copy the run-time module as well.

BASIC Run-Time Libraries

You may choose to link with a run-time library instead of with a stand-alone library. If you compile and then link your program with a run-time library, the resulting executable file can only be run in the presence of a run-time module. This module contains code that implements the BASIC language.

Using a run-time module and library provides the following advantages:

- The executable file is much smaller. If several programs are kept on the disk, considerable space is saved.
- Unnamed **COMMON** variables and open files are preserved across **CHAIN** statements. This can be valuable in systems of programs that use shared data.
- Run-time modules reside in memory, so they do not need to be reloaded for each program in a system of chained programs.

Run-time modules use the following naming conventions for real and protected modes and run-time libraries:

BRT70 *float string mode*.EXE (Real mode)

BRT70 *float string mode*.DLL (Protected mode)

Run-time libraries use the following naming convention:

BRT70 *float string mode*.LIB

The possible values for each variable are the same as for stand-alone libraries (see Table 17.1). For example, if you specify the emulator math, far strings, and real mode options during the setup operation, your program would use the BRT70EFR.EXE run-time module and the BRT70EFR.LIB run-time library.

In addition to the standard run-time module, Microsoft BASIC also lets you embed your own routines into the run-time module to create a custom run-time module. To do this, you write and compile your source modules; create a file called an export list to define object files, routines, and libraries you wish to add to the run-time module; invoke the BUILDRTM utility to create the custom run-time module and support files; and, finally, link the object files for your application with the support files for your custom run-time module. These steps are described in detail in Chapter 21, "Building Custom Run-Time Modules."

Add-On Libraries

Microsoft BASIC provides several additional libraries. These libraries provide additional support for various types of functions that you may want to include in your BASIC programs.

The following libraries are supplied as object-module libraries:

Type of library	Name
Financial functions	FINANCER.LIB (emulator, real mode) FINANCEP.LIB (emulator, protected mode) FINANCAR.LIB (alternate math, real mode) FINANCAP.LIB (alternate math, protected mode)
Date/time functions	DTFMTER.LIB (emulator, real mode) DTFMTEP.LIB (emulator, protected mode) DTFMTAR.LIB (alternate math, real mode) DTFMTAP.LIB (alternate math, protected mode)

Note Quick library versions of these libraries are provided for the emulator in real mode (the only valid options for use in QBX).

For information about routines in the add-on libraries, see the *BASIC Language Reference*.

BASIC Toolbox Files

BASIC toolbox files are BASIC source files that you can use to build object-module or Quick libraries. Microsoft BASIC supplies the following toolbox files:

Type of file	Name
Matrix	MATB.BAS
Presentation graphics	CHRTB.BAS
Font	FONTB.BAS
Menu	MENU.BAS
Mouse	MOUSE.BAS
Window	WINDOW.BAS
Support	GENERAL.BAS

After compiling these files into object files, you can use LIB to create an object-module library or LINK to create a Quick library. See Chapter 18, "Using LINK and LIB," for details on how to create an object-module library or Chapter 19, "Creating and Using Quick Libraries," for details on how to create a Quick library.

For information about the routines found in these libraries, see the *BASIC Language Reference*.

Quick Libraries (.QLB)

Microsoft BASIC supports another type of library called a Quick library. This type of library can only be used in the QBX environment—it can't be linked with object modules to create an executable file. Quick libraries normally have the .QLB filename extension.

Quick libraries contain one or more procedures that can be loaded and made available to your BASIC programs in the QBX environment. A Quick library can contain procedures written in BASIC or other Microsoft languages such as C. Procedures in a Quick library behave like QBX's own statements. Just like BASIC statements, Quick library procedures can be executed directly from the Immediate window or called from your BASIC program. This makes it easy to test them before using them in other programs.

Advantages of Quick Libraries

There are several reasons why you might want to use Quick libraries. Quick libraries facilitate program development and maintenance. As development progresses on a project and modules become stable components of your program, you can add them to a Quick library, then set aside the source files for the original modules until you want to improve or maintain those source files. Thereafter you can load the library along with QBX, and your program has instant access to all procedures in the library.

If you develop programs with others, Quick libraries make it easy to update a pool of common procedures. If you wish to offer a library of original procedures for commercial distribution, all QBX programmers will be able to use them immediately to enhance their own work. You could leave your custom Quick library on a bulletin board for others to try before purchasing. Because Quick libraries contain no source code and can only be used within the QBX programming environment, your proprietary interests are protected while your marketing goals are advanced.

BASIC procedures within Quick libraries represent code compiled with the command-line compiler. These procedures share significant characteristics with executable files. For example, executable files and Quick libraries perform floating-point arithmetic faster than the same calculations performed within the QBX environment.

Note Quick libraries have the same function as user libraries in QuickBASIC versions 2.0 and 3.0. However, you cannot load a user library as a Quick library. You must recreate the library from the original source code, as described in the following section.

Making Quick Libraries

If your source modules are written in BASIC, you can create Quick libraries from either QBX or the command line. However, if you have source modules written in languages besides BASIC, you must create the Quick library from the command line.

From QBX, load desired BASIC source modules into the environment and choose the Make Library option from the Run menu to build the library. QBX automatically compiles your BASIC source modules, then creates a Quick library (.QLB) and a parallel object-module library (.LIB) as shown in Figure 17.1.

Figure 17.1 Making a Quick Library from QBX

To make a Quick library from the command line, you must compile all source modules using the appropriate compiler (or assembler), then invoke LINK to create the Quick library. You should also create a parallel object-module library using the LIB.

Since Quick libraries are managed by LINK, you can't manipulate objects in it as with object-module libraries. To remove, add, or modify modules in a Quick library, you must work with its parallel object-module library.

Instructions for creating, modifying, and using Quick libraries are found in Chapter 19, "Creating and Using Quick Libraries."

Overview of Linking

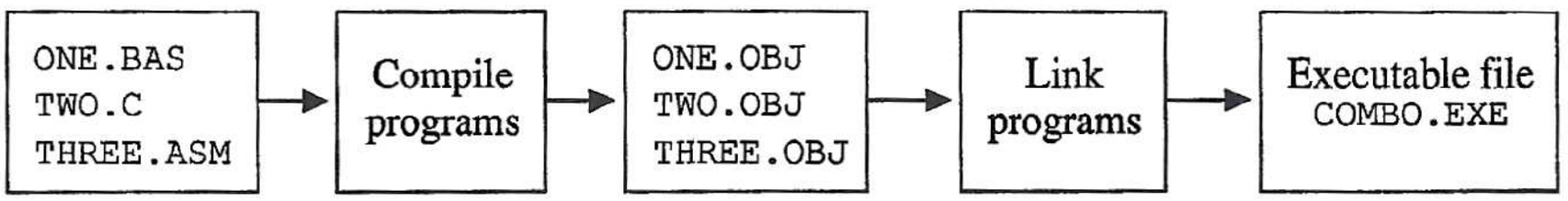
Linking is the process of combining libraries and other object files to create an executable file or a Quick library. To do this, Microsoft BASIC provides the LINK utility. To create a Quick library, LINK must be invoked with the /Q (Quick library) option. To create an executable file, LINK is invoked without the /Q option.

LINK can be invoked from either QBX or from the command line. From QBX, LINK is invoked automatically when you select either the Make EXE or Make Library option from the Run menu. From the command line, LINK is invoked by typing LINK on the command line followed by command-line arguments and options.

Using LINK to Create Executable Files

LINK can be used to create an executable file from object modules and libraries. You can write source modules in any standard Microsoft language such as BASIC, C, or MASM. Then compile the programs using the appropriate compiler to produce an object file (.OBJ). Finally, use LINK to combine the object modules with appropriate libraries.

For example, say you have a written a BASIC program ONE.BAS, which calls a C program named TWO.C and an assembly language program named THREE.ASM. You would first compile each program separately, and then link the resulting object files to produce an executable file:

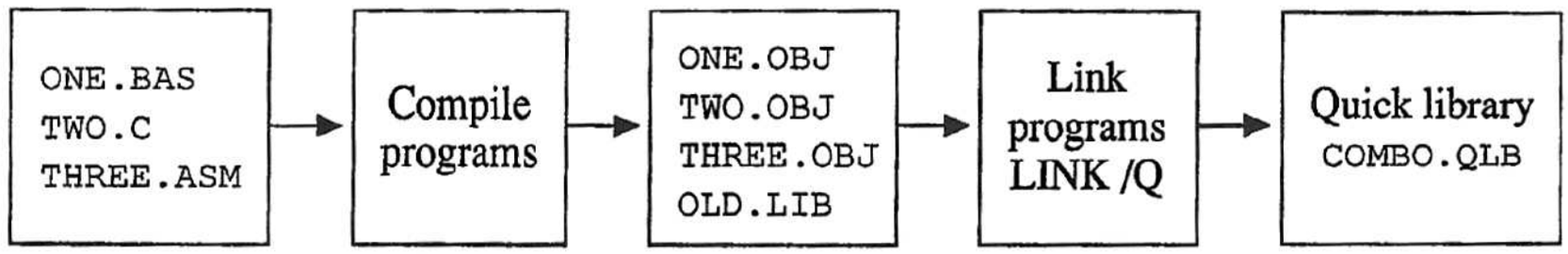


The file COMBO.EXE can be run from the DOS command prompt. LINK also supports other features such as producing a map file and reading information from a module definition file (OS/2 programs). For details on using LINK, see Chapter 18, "Using LINK and LIB."

Using LINK to Create Quick Libraries

If you specify the /Q option, the LINK will create a Quick library instead of an executable file. You use object modules such as compiled source code or libraries as input to LINK.

For example, the files ONE.BAS, TWO.C, and THREE.ASM could be compiled and then linked along with the library OLD.LIB to create a Quick library by specifying the /Q option when you invoke LINK:



For more information about creating Quick libraries, see Chapter 19, "Creating and Using Quick Libraries."

Linking with Stub Files

Microsoft BASIC provides several special-purpose object files with which you can link to minimize the size of your executable file. These files, called "stub files," cause LINK to exclude (or in some cases, include a smaller version of) code that would normally be placed in your executable file.

For example, if your program does not need editor support, you could link your program with the NOEDIT.OBJ stub file. LINK would then include code that replaces the editor used with the **INPUT** and **LINE INPUT** statements. Keep in mind that this process is appropriate only for programs compiled with the /O option of BC or when creating a custom run-time module.

The stub files supplied with Microsoft BASIC and how to link with them are discussed in Chapter 18, "Using LINK and LIB."

Linking to Create Overlays

You can direct LINK to create an overlaid version of a program. In an overlaid version of a program, specified parts of the program (known as "overlays") are loaded only if and when they are needed. These parts share the same space in memory. Only code is overlaid; data is never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to read and reread the code from disk or Expanded Memory Specification (EMS) into conventional memory. Specifying overlays can be useful if you have compiled a program that is too large to load into memory. You can have only those portions of code loaded into memory that are currently needed. Overlays are discussed in Chapters 15, "Optimizing Program Size and Speed," and 18, "Using LINK and LIB."

Chapter 18
Using LINK and LIB

This chapter describes the syntax and usage of the Microsoft Library Manager (LIB) and the Microsoft Segmented-Executable Linker (LINK). The syntax to invoke each utility is provided, as well as descriptions of command-line arguments and options. For an overview of linking and libraries, see Chapter 17, "About Linking and Libraries."

Invoking and Using LIB

The Microsoft Library Manager (LIB) helps you create and maintain object-module libraries. An object-module library is a collection of separately compiled or assembled object files combined into a single file. Object-module libraries provide a convenient source of commonly used routines. A program that calls library routines is linked with the library to produce the executable file. Only modules containing the necessary routines, not all library modules, are linked into the executable file.

Library files are usually identified by their .LIB extension, although other extensions are allowed. In addition to accepting DOS object files and library files, LIB can read the contents of 286 XENIX archives and Intel-style libraries and combine their contents with DOS libraries.

You can use LIB for the following tasks:

- Create a new library file.
- Add object files or the contents of a library to an existing library.
- Delete library modules.
- Replace library modules.
- Copy library modules to object files.

While the Microsoft BASIC Setup program creates libraries during installation, you will need to run LIB if you want to create new libraries (for example, from BASIC object modules) or if you want to modify the contents of an existing library. This lets you create custom libraries containing only those routines that you want.

To invoke LIB, type the LIB command on the DOS command line. You can specify the input required in one of three ways:

- Type it on the command line after the LIB command.
- Respond to prompts.
- Specify a file containing responses to prompts (called a "response file").

This section describes how to specify input to LIB on the command line. To use prompts or a response file to specify input, see the sections "LIB Prompts" or "LIB Response File" later in this chapter.

The command-line syntax for LIB is as follows:

LIB *oldlibrary* [[*options*]] [[*commands*]] [[, [[*listfile*]] [[, [[*newlibrary*]]]]]] [[;]]

Descriptions of the fields are summarized in the following table:

Field	Description
oldlibrary	Name of the existing library that will be operated upon
options	Options to the LIB command
commands	Command symbols for manipulating modules
listfile	Name of a cross-reference listing file to be generated
newlibrary	Name of the modified library created by LIB

The individual fields are discussed in greater detail in the sections that follow.

Type a semicolon (;) after any field except the *oldlibrary* field to tell LIB to use the default responses for the remaining fields. The semicolon should be the last character on the command line. Typing a semicolon after the *oldlibrary* field causes LIB to perform a consistency check on the library — no other action is performed. LIB displays any consistency errors it finds and returns to the operating-system level (see the section "Consistency Check" later in this chapter for more information).

You can terminate the library session at any time and return to the operating system by pressing Ctrl+C.

Old Library File

Use the *oldlibrary* field to specify the name of the library to be created, modified, or operated upon. LIB assumes that the filename extension is .LIB, so if your library file has the .LIB extension, you can omit it. Otherwise, include the extension. You must give LIB the path of a library file if it is in another directory or on another disk.

There is no default for the *oldlibrary* field. This field is required and LIB issues an error message if you do not give a filename. If the library you name does not exist, LIB displays the following prompt:

```
Library does not exist. Create? (y/n)
```

Type **Y** to create the library file, or **N** to terminate the session. This message does not appear if a command, a comma, or a semicolon immediately follows the library name.

Consistency Check

If you type a library name and follow it immediately with a semicolon (;), LIB only performs a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. No changes are made to the library. It usually is not necessary to perform consistency checks because LIB automatically checks object files for consistency before adding them to the library. LIB prints a message if it finds an invalid object module; no message appears if all modules are intact.

Creating a Library File

To create a new library file, give the name of the library file you want to create in the *oldlibrary* field of the command line (or at the "Library name" prompt). LIB supplies the .LIB extension, if needed.

If the name of the new library file is the same as the name of an existing library file, LIB assumes that you want to change the existing file. If the name of the new library file is the same as the name of a file that is not a library, LIB issues an error message.

When you give the name of a file that does not currently exist, LIB displays the following prompt:

```
Library does not exist. Create? (y/n)
```

Type **Y** to create the file, or **N** to terminate the library session. This message does not appear if the name is followed immediately by a command, a comma, or a semicolon.

You can specify a page size for the library by specifying the /PA:*number* option when you create the library. The default page size is 16 bytes.

Once you have given the name of the new library file, you can insert object modules into the library by using the add-command symbol (+) (described under "Commands" later in this chapter).

Examples

The following example causes LIB to perform a consistency check of the library file GRAPHIC.LIB:

```
LIB GRAPHIC;
```

The following example tells LIB to perform a consistency check of the library file GRAPHIC.LIB and to create SYMBOLS.LST, a cross-reference-listing file:

```
LIB GRAPHIC ,SYMBOLS.LST;
```

Options

LIB has six options, which are listed in the following table:

Option	Description
/HELP	Display information about LIB syntax and options.
/I	Ignore case when comparing symbols.
/NOD	Prevent extended dictionary from being generated.
/NOI	Do not ignore case when comparing symbols.
/NOLOGO	Do not display sign-on banner.
/PA:*number*	Specify library page size.

Specify options on the command line following the required library filename and preceding any commands.

Ignoring Case of Symbols (/I)

The /I option tells LIB to ignore case when comparing symbols, which is the default. Use this option when you are combining a library that is case sensitive (was created with the /NOI option) with others that are not case sensitive. The resulting library will not be case sensitive. The /NOI option is described later in this section.

No Extended Dictionary (/NOE)

The /NOE option tells LIB not to generate an extended dictionary. The extended dictionary is an extra part of the library that helps LINK process libraries faster.

Use the /NOE option if you get the error message `Insufficient memory` or `No more virtual memory`, or if the extended dictionary causes problems with LINK (that is, if you receive the message `Symbol multiply defined`). For more information on how LINK uses the extended dictionary, see the description of the /NOE option for LINK.

Using Case-Sensitive Symbols (/NOI)

The /NOI option tells LIB not to ignore case when comparing symbols; that is, /NOI makes LIB case sensitive. By default, LIB ignores case. Using this option allows symbols that are the same except for case, such as Spline and SPLINE, to be put in the same library.

Note that when you create a library with the /NOI option, LIB "marks" the library internally to indicate that /NOI is in effect. Earlier versions of LIB did not mark libraries in this way. If you combine multiple libraries and any one of them is marked /NOI, then /NOI is assumed to be in effect for the output library.

Specifying Page Size (/PA:number)

The /PA option specifies the library page size of a new library or changes the library page size of an existing library. The *number* specifies the new page size. It must be an integer value representing a power of two between the values 16 and 32,768.

A library's page size affects the alignment of modules stored in the library. Modules in the library are always aligned to start at a position that is a multiple of the page size (in bytes) from the beginning of the file. The default page size for a new library is 16 bytes; for an existing library, the default is its current page size. Because of the indexing technique used by LIB, a library with a large page size can hold more modules than a library with a smaller page size. For each module in the library, however, an average of *number* / 2 bytes of storage space is wasted. In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

Another consequence of the indexing technique is that the page size determines the maximum possible size of the library file. Specifically, this limit is *number* * 65,536. For example, /PA:16 means that the library file must be smaller than 1 megabyte (16 * 65,536 bytes).

Commands

LIB can perform a number of library-management functions, including creating a library file, adding an object file as a module to a library, deleting a module from a library, replacing a module in the library file, copying a module to a separate object file, and moving a module out of a library and into an object file.

The *commands* field allows you to specify the command symbols for manipulating modules. In this field, type a command symbol followed immediately by a module name or the name of an object file. The command symbols are the following:

Symbol	Action
+	Adds an object file or library to the library.
–	Deletes a module from the library.
–+	Replaces a module in the library.
*	Copies a module from the library to an object file.
–*	Moves a module (copies the module and then deletes it).

Each of these commands is described in the following sections. Note that LIB does not process commands in left-to-right order; it uses its own precedence rules for processing (described in the next section). You can specify more than one operation in the *commands* field, in any order. LIB makes no changes to *oldlibrary* if you leave this field blank.

Order of Processing

For each library session, LIB reads and interprets commands in the following order. It determines whether a new library is being created or an existing library is being examined or modified.

1. LIB processes any deletion and move commands.

   LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

2. LIB processes any addition commands.

   Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or move commands, a new library file would be created in the addition stage by copying the original library file.)

How LIB Processes Commands

As LIB carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. When you link a library with other object files, LINK uses the library index to search the library.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. Therefore, if you press Ctrl+C to terminate the session, you do not lose your original library. Because of this, when you run LIB, you must make sure your disk has enough space for the original library file and the copy.

Once an object file is incorporated into a library, it becomes an "object module." An object file has a full path, including a drive designation, directory path, and filename extension (usually .OBJ) object modules have only a name. For example, B:\RUN\SORT.OBJ is an object filename, while SORT is an object module name.

Add Command (+)

Use the add-command symbol (+) to add an object module to a library. Give the name of the object file to be added, without the .OBJ extension, immediately following the plus sign.

LIB uses the base name of the object file as the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is CURSOR.

Object modules are always added to the end of a library file.

You can also use the plus sign to combine two libraries. When you give a library name following the plus sign, a copy of the contents of that library is added to the library file being modified. You must include the .LIB extension when you give a library filename. Otherwise, LIB uses the default .OBJ extension when it looks for the file. If both libraries contain a module with the same name, LIB ignores the second module of that name. For information on replacing modules, see the description of the replace-command symbol (–+) found later in this chapter.

LIB adds the modules of the library to the end of the library being changed. Note that the added library still exists as an independent library because LIB copies the modules without deleting them.

In addition to allowing DOS libraries as input, LIB also accepts 286 XENIX archives and Intel-format libraries. Therefore, you can use LIB to convert libraries from either of these formats to the DOS format.

Examples The following example uses the add-command symbol (+) to instruct LIB to add the file STAR to the library GRAPHIC.LIB:

```
LIB GRAPHIC +STAR;
```

The semicolon at the end of the preceding command line causes LIB to use the default responses for the remaining fields. As a result, no listing file is created and the original library file is renamed GRAPHIC.BAK. The modified library is GRAPHIC.LIB.

The following example adds the file FLASH.OBJ to the library MAINLIB.LIB:

```
LIB MAINLIB +FLASH;
```

The following example adds the contents of the library TRIG.LIB to the library MATH.LIB. The library TRIG.LIB is unchanged after this command is executed.

```
LIB MATH +TRIG.LIB;
```

Delete Command (–)

Use the delete-command symbol (–) to delete an object module from a library. After the minus sign, give the name of the module to be deleted. Module names do not have paths or extensions. The contents of the deleted library are copied to another file having the same filename except with a .BAK extension.

Example The following example deletes the module FLASH from the library MAINLIB.LIB:

```
LIB MAINLIB -FLASH;
```

Replace Command (–+)

Use the replace-command symbol (–+) to replace a module in a library. Following the symbol, give the name of the module to be replaced. Module names do not have paths or extensions.

To replace a module, LIB first deletes the existing module, then appends an object file that has the same name as the module. The object file is assumed to have the .OBJ extension and to reside in the current directory; if not, give the object filename with an explicit extension or path.

Example This command replaces the module FLASH in the MAINLIB.LIB library with the contents of FLASH.OBJ from the current directory. Upon completion of this command, the file FLASH.OBJ still exists and the FLASH module is updated in MAINLIB.LIB.

```
LIB MAINLIB -+FLASH;
```

Copy Command (*)

Use the copy-command symbol (*) followed by a module name to copy a module from the library into an object file of the same name. The module remains in the library. When LIB copies the module to an object file, it adds the .OBJ extension to the module name and places the file in the current directory.

Example The following example copies the module FLASH from the MAINLIB.LIB library to a file called FLASH.OBJ in the current directory. Upon completion of this command, MAINLIB.LIB still contains the module FLASH.

```
LIB MAINLIB *FLASH;
```

Move Command (–*)

Use the move-command symbol (–*) to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.

Example The following example moves the module FLASH from the MAINLIB.LIB library to a file called FLASH.OBJ in the current directory. Upon completion of this command, MAINLIB.LIB no longer contains the module FLASH.

```
LIB MAINLIB -*FLASH;
```

This following example instructs LIB to move the module JUNK from the library GRAPHIC.LIB to an object file named JUNK.OBJ. The module JUNK is removed from the library in the process.

```
LIB GRAPHIC -*JUNK *STAR, ,SHOW
```

In the preceding example, the module STAR is copied from the library to an object file named STAR.OBJ; the module remains in the library. No cross-reference listing file is produced. The revised library is named SHOW.LIB. It contains all the modules in GRAPHIC.LIB except JUNK, which was removed by using the move-command symbol (–*). The original library, GRAPHIC.LIB, remains unchanged.

Cross-Reference Listing File

The *listfile* field allows you to specify a filename for a cross-reference listing file. You can give the listing file any name and any extension. To create it outside your current directory, supply a path. Note that LIB does not assume any defaults for this field on the command line. If you do not specify a name for the file, the file is not created.

A cross-reference listing file contains the following two lists:

- An alphabetical list of all public symbols in the library.

  Each symbol name is followed by the name of the module in which it is defined. The following example output shows that the public symbol `ADD` is contained in the module `junk` and the public symbols `CALC`, `MAKE`, and `ROLL` are contained in the module `dice`:

```
ADD..............junk CALC.............dice
MAKE.............dice ROLL.............dice
```

- A list of the modules in the library.

  Under each module name is an alphabetical listing of the public symbols defined in that module. The following example output shows that the module dice contains the public symbols `CALC`, `MAKE`, and `ROLL` and the module `junk` contains the public symbol `ADD`:

```
dice Offset: 00000010H Code and data size: 621H
 CALC MAKE ROLL
junk Offset: 00000bc0H Code and data size: 118H
 ADD
```

New Library

If you specify a name in the *newlibrary* field, LIB gives this name to the modified library it creates. This optional field is only used if you specify commands to change the library.

If you leave this field blank, the original library is renamed with a .BAK extension and the modified library receives the original name.

LIB Prompts

If you type **`LIB`** at the DOS command line, the library manager prompts you for the input it needs by displaying the following four messages, one at a time:

```
Library name:
Operations:
List file:
Output library:
```

The input for each prompt corresponds to each field of the LIB command. See the previous sections for descriptions of each LIB command field.

LIB waits for you to respond to each prompt before printing the next prompt. If you notice that you have entered an incorrect response to a previous prompt, press Ctrl+C to exit LIB and begin again.

Extending Lines

If you have many operations to perform during a library session, use the ampersand symbol (&) to extend the operations line. Type the ampersand symbol after the name of an object module or object file; do not put the ampersand between a command symbol and a name.

The ampersand causes LIB to display the Operations prompt again, allowing you to specify more operations.

Default Responses

Press the Enter key to choose the default response for the current prompt. Type a semicolon (;) and press Enter after any response except "Library name" to select default responses for all remaining prompts.

The following list shows the defaults for LIB prompts:

Prompt	Default
Operations	No operation; no change to library file
List file	NUL; no listing file is produced
Output library	The current library name

LIB Response File

Using a response file lets you conduct the library session without typing responses to prompts at the keyboard. To run LIB with a response file, you must first create the response file. Then type the following at the DOS command line:

LIB @*responsefile*

The *responsefile* is the name of a response file. Specify a path if the response file is not in the current directory.

You can also enter @*responsefile* at any position on a command line or after any of the prompts. The input from the response file is treated exactly as if it had been entered on a command line or after the prompts. A new-line character in the response file is treated the same as pressing the Enter key in response to a prompt.

A response file uses one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would use responses typed on the keyboard. You can type an ampersand (&) at the end of the response to the Operations prompt, for instance, and continue typing operations on the next line.

When you run LIB with a response file, the prompts are displayed with the responses from the response file. If the response file does not contain responses for all the prompts, LIB uses the default responses.

Example Assume that a response file named RESPONSE in the directory B:\PROJ contains the following lines:

```
GRAPHIC
+CIRCLE+WAVE-WAVE*FLASH
GRAPHIC.LST
```

If you invoke LIB with the following command line, then LIB deletes the module WAVE from the library GRAPHIC.LIB, copies the module FLASH into an object file named FLASH.OBJ, appends the object files CIRCLE.OBJ and WAVE.OBJ as the last two modules in the library, and creates a cross-reference listing file named GRAPHIC.LST.

```
LIB @B:\PROJ\RESPONSE
```

Invoking and Using LINK

This section describes how to use the Microsoft Segmented-Executable Linker (LINK). LINK allows you to link object files with appropriate libraries to create an executable file or Quick library.

You can invoke LINK in several ways. If you are working in the QBX environment and have selected the Make EXE file option from the Run menu, LINK is automatically called after your program is compiled by BASIC. If you are compiling and linking your program from the command line, you must invoke LINK separately after compiling your program with BC. You can also invoke LINK and specify LINK options from within a MAKEFILE (described in Chapter 20, "Using NMAKE").

Regardless of how you invoke LINK, you may press Ctrl+C at any time to terminate a LINK operation and exit to the operating system.

You can specify the input required for the LINK command in one of three ways:

- By placing it on the command line.
- By responding to prompts.
- By specifying a file containing responses to prompts. This type of file is known as a "response file."

This section describes how to invoke LINK from the command line. For information about responding to prompts or using a response file, see the sections "LINK Prompts" and "LINK Response File" later in this chapter.

The command-line syntax for the LINK command is as follows:

LINK [[*options*]] *objfiles* [[, [[*exefile*]] [[, [[*mapfile*]] [[, [[*libraries*]] [[, [[*deffile*]]]]]]]]]] [[;]]

Descriptions of the fields are summarized in the following table:

Field	Description
options	Options to the LINK command
objfiles	Names of the object files to be linked
exefile	Name of the executable file generated
mapfile	Name of the map file generated
libraries	Name(s) of one or more libraries you want linked with *objfiles*
deffile	Name of a module-definition file

Note

A module-definition file is needed only for OS/2 protected mode and Microsoft Windows programs (not compatible with BASIC programs); however, this prompt is still issued when you are linking DOS programs.
The command line fields are described fully in the sections that follow.

A comma must separate each command-line field from the next. You may omit the text from any field (except the required *objfiles*), but you must include the comma. A semicolon may end the command line after any field, causing LINK to use defaults for the remaining fields.

The following table shows the default values used by LINK:

Field	Default
options	The name of the first object file submitted for the "Object Modules" prompt with the .EXE extension replacing the .OBJ extension
mapfile	The special filename NUL.MAP, which tells LINK *not* to create a map file
libraries	The default libraries encoded in the object files (see "How LINK Searches for Libraries" later in this chapter)
deffile	The NUL.DEF file, which tells LINK not to use a module definition file

Examples

The following example causes LINK to load and link the object modules SPELL.OBJ, TEXT.OBJ, DICT.OBJ, and THES.OBJ, and to search for unresolved references in the library XLIB.LIB as well as in the default library created during setup.

```
LINK SPELL+TEXT+DICT+THES, ,SPELLIST, XLIB.LIB;
```

By default, the executable file produced by LINK is named SPELL.EXE. LINK also produces a map file, SPELLIST.MAP. The semicolon at the end of the line tells NMAKE to accept the default module-definition file (NUL.DEF).

The following example produces a map file named SPELL.MAP because a comma appears as a placeholder for the map file specified on the command line:

```
LINK SPELL,,;
```

The following example does not produce a map file because commas do not appear as placeholders for the map file specified:

```
LINK SPELL,;
LINK SPELL;
```

The following example causes LINK to link the three files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ into an executable file. A map file named MAIN.MAP is also produced:

```
LINK MAIN+GETDATA+PRINTIT, , MAIN;
```

Default Filename Extensions

You can use any combination of uppercase and lowercase letters for the filenames you specify on the LINK command line or give in response to the LINK command prompts. If you specify filenames without extensions, LINK uses the following default filename extensions:

Type of file	Default extension
Object	.OBJ
Executable	.EXE or .QLB
Map (or "listing")	.MAP
Library	.LIB
Definitions	.DEF

You can override the default extension for a particular command-line field or prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period.

Choosing Defaults

If you include a comma (to indicate where a field would be) but do not put a filename before the comma, then LINK selects the default for that field. However, if you use a comma to include the *mapfile* field (but do not include a name), then LINK creates a map file. This file has the same base name as the executable file. Use NUL for the map filename if you do not want to produce a map file.

You can also select default responses by using a semicolon (;). The semicolon tells LINK to use the defaults for all remaining fields. Anything after the semicolon is ignored. If you do not give all filenames on the command line or if you do not end the command line with a semicolon, LINK prompts you for the files you omitted. Descriptions of these prompts are given in the following section.

The following table summarizes LINK's defaults for each field:

Field	Default
exefile	Creates a file with the base name of the first object file and a .EXE extension.
mapfile	Does not create a map file unless you include the *mapfile* field. The field may be empty, as in the following command line: `LINK MYFILE YOURFILE, OURFILE, ;` If you include the field but not a filename, LINK creates a map file with the base name of the executable file and the .MAP extension. Thus the example creates a map file named OURFILE.MAP.
libraries	Searches only the default libraries specified in the object files.
deffile	Does not accept input from a module-definition file.

If you do not specify a drive or directory for a file, LINK assumes that the file is on the current drive and directory. If you want LINK to create files in a location other than the current drive and directory, you must specify the new drive and directory for each such file on the command line.

LINK Options

The *linkoptions* field contains command-line options to LINK. You may specify command-line options after any field, but before the comma that terminates the field. You do not have to give any options when you run LINK. See the sections that follow for individual descriptions of command-line LINK options.

This section explains how to use LINK options to specify and control the tasks performed by LINK. When you use the LINK command line to invoke LINK, you may put options at the end of the line or after individual fields on the line. Options, however, must immediately precede the comma that separates each field from the next.

If you respond to the individual prompts for the LINK command, you may specify LINK options at the end of any response. When you use more than one option, you can either group the options at the end of a single response or distribute the options among several responses. Every option must begin with the slash character (/) or a dash (-), even if other options precede it on the same line.

In a response file, options may appear on a line by themselves or after individual response lines.

Abbreviations

Because LINK options are named according to their functions, some of their names are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique so that LINK can determine which option you want. The minimum legal abbreviation for each option is indicated in the description of that option listed in the sections that follow.

Abbreviations must begin with the first letter of the name and must be continuous through the last letter typed. No spaces or transpositions are allowed. Options may be entered in uppercase or lowercase letters.

Numeric Arguments

Some LINK options take numeric arguments. A numeric argument can be any of the following:

- A decimal number from 0 to 65,535.
- An octal number from 00 to 0177777. A number is interpreted as octal if it starts with 0. For example, the number "10" is interpreted as a decimal number, but the number "010" is interpreted as an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0X0 to 0XFFFF. A number is interpreted as hexadecimal if it starts with 0X. For example, "0X10" is a hexadecimal number, equivalent to 16 in decimal.

LINK Environment Variable

You can use the LINK environment variable to cause certain options to be used each time you link. LINK checks the environment variable for options if the variable exists.

LINK expects to find options listed in the variable exactly as you would type them on the command line. It does not accept any other arguments; for instance, including filenames in the environment variable causes the error message `Unrecognized option name`.

Each time you link, you can specify other options in addition to those in the LINK environment variable. If you enter the same option on the command line and in the environment variable, LINK ignores the redundant option. If the options conflict, however, the command-line option overrides the effect of the environment-variable option. For example, the command-line option /SE:512 cancels the effect of the environment-variable option /SE:256.

Note Unless you override it on the command line, the only way to prevent an option in the environment variable from being used is to reset the environment variable itself.

Examples In the following example, the file TEST.OBJ is linked with the options /NOI, /SE:256, and /CO:

```
SET LINK=/NOI /SE:256 /CO
LINK TEST;
```

In the next example, the file PROG.OBJ is then linked with the option /NOD, in addition to /NOI, /SE:256, and /CO (note that the second /CO option is ignored):

```
LINK /NOD /CO PROG;
```

Valid LINK Options

LINK provides many options that can be used to link programs written in several Microsoft languages. While most are valid for BASIC programs, several should not be used. Table 18.1 lists those options that are valid for BASIC programs.

Table 18.1 LINK Options Supported with BASIC

Option	Description
/A[LIGNMENT]:*size*	Aligns segment data according to *size*.
/BA[TCH]	Prevents LINK from prompting for path when it cannot find specified libraries or object files.
/CO[DEVIEW]	Prepares program for debugging with CodeView.
/DO[SSEG]	Forces a special ordering on segments. Not for use in custom run-time modules.
/E[XEPACK]	Packs executable files during linking.
/F[ARCALLTRANSLATION]	Optimizes far calls to procedures in the same segment as the caller. Use in conjunction with /PACKCODE.
/HE[LP]	Lists LINK options to standard output.
/INC[REMENTAL]	Prepares a program for incremental linking with ILINK.

Table 18.1 *Continued*

Option	Description
/INF[[ORMATION]]	Displays LINK process information.
/LI[[NENUMBERS]]	Includes line numbers in the map file.
/M[[AP]]	Lists public symbols defined in the object file.
/NOD[[EFAULTLIBRARYSEARCH]]	Ignores default libraries during linking.
/NOE[[XTDICTIONARY]]	Ignores extended dictionary during linking.
/NOF[[ARCALLTRANSLATION]]	Disables far-call optimization (/FARCALLTRANSLATION).
/NOI[[GNORECASE]]	Preserves case sensitivity. Not for use in OS/2 custom run-time modules.
/NOL[[OGO]]	Suppresses the sign-on logo.
/NON[[ULLDOSSEG]]	Works the same as the /DOSSEG option, except that no null bytes are inserted at the beginning of the _TEXT segment.
/NOP[[ACKCODE]]	Disables segment packing (/PACKCODE).
/O[[VERLAYINTERRUPT]]:*number*	Specifies an interrupt number other than 0x3F for passing control to overlays.
/PAC[[KCODE]]	Packs contiguous code segments. Do not use this option if you have overlays.
/PACKD[[ATA]]	Packs contiguous data segments.
/PADC[[ODE]] :*padsize*	Adds filler bytes to the end of each code segment. For use with ILINK.
/PADD[[ATA]]:*padsize*	Adds filler bytes to the end of each data segment. For use with ILINK.
/PAU[[SE]]	Causes LINK to pause before writing the executable file to disk.
/PM[[TYPE]]:*type*	Specifies OS/2 program type. See the full description later in this chapter for restrictions.
/Q[[UICKLIBRARY]]	Produces Quick library for use with QBX.
/SE[[GMENTS]]:*number*	Sets maximum number of segments that LINK allows a program to have.
/W[[ARNFIXUP]]	Issues fixup warnings. Not for use with custom run-time modules. For BASIC programs, must be used in conjunction with /NOPACKCODE.

Complete descriptions for each of these options are found later in this chapter.

Invalid LINK Options

Not all options of the LINK command are suitable for use with BASIC programs. Table 18.2 lists options that do not have an effect or that should not be used with BASIC programs.

Table 18.2 LINK Options not Supported with BASIC

Option	Action
/CP[ARMAXALLOC]:*number*	Sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. Although you can use this option, it has no effect because, while it is running, your BASIC program controls memory.
/DS[ALLOCATE]	Loads all data starting at the high end of the default data segment.
/HI[GH]	Places the executable file as high in memory as possible.
/ST[ACK]:*number*	Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. The standard BASIC library sets the default stack size to 3K for DOS and 3.5K for OS/2. BASIC programs should use the **STACK** statement or function instead.
/T[INY]	Produces .COM files for small programs where DOS load time is an issue.

Note The /DS and /HI options are suitable only for object files created by the MASM.

Aligning Segment Data (/A:size)

This option directs LINK to align segment data in the executable file along with the boundaries specified by *size*. The *size* argument must be a power of two. For example, /A:16 indicates an alignment boundary of 16 bytes. The default alignment for OS/2 application and dynamic-link segments is 512. This option is used for linking Windows applications or protected-mode programs.

Running in Batch Mode (/BA)

By default, LINK prompts you for a new path whenever it cannot find a library that it has been directed to use. It also prompts you if it cannot find an object file that it expects to find on a removable disk. If you use the /BA option, however, LINK does not prompt you for any libraries or object files that it cannot find. Instead, LINK generates an error or warning message, if appropriate. In addition, when you use /BA, LINK does not display its copyright banner, nor does it echo commands from response files. This option does not prevent LINK from prompting for command-line arguments. You can prevent such prompting only by using a semicolon on the command line or in a response file.

Using this option may result in unresolved external references. It is intended primarily for use with batch or NMAKE files that link many executable files with a single command and to prevent LINK operation from halting.

Note In earlier versions of LINK, the /BATCH option was abbreviated to /B.

Preparing for Debugging (/CO)

The /CO option is used to prepare for debugging with the Microsoft CodeView window-oriented debugger. This option tells LINK to prepare a special executable file containing symbolic data and line-number information.

Object files linked with the /CO option must first be compiled with the /Zi option, which is described in Chapter 16, "Compiling With BC."

You can run this executable file outside the CodeView debugger; the extra data in the file is ignored. To keep file size to a minimum, however, use the special-format-executable file only for debugging; then you can link a separate version without the /CO option after the program is debugged.

Ordering Segments (/DO)

The /DO option forces a special ordering on segments. This option is automatically enabled by a special object-module record in Microsoft BASIC libraries. If you are linking to one of these libraries, then you do not need to specify this option.This option is also enabled by assembly modules that use the MASM directive .DOSSEG.

The /DO option forces segments to be ordered as follows:

1. All segments with a class name ending in CODE
2. All other segments outside DGROUP
3. DGROUP segments, in the following order:
   a. Any segments of class BEGDATA (this class name reserved for Microsoft use)
   b. Any segments not of class BEGDATA, BSS, or STACK
   c. Segments of class BSS
   d. Segments of class STACK

When the /DO option is in effect LINK initializes two special variables as follows:

```
_edata = DGROUP : BSS
_end = DGROUP : STACK
```

The variables `_edata` and `_end` have special meanings for the Microsoft C and FORTRAN compilers, so it is not wise to give these names to your own program variables. Assembly modules can reference these variables but should not change them.

Packing Executable Files (/E)

The /E option directs LINK to remove sequences of repeated bytes (typically null characters) and to optimize the load-time-relocation table before creating the executable file. (The load-time-relocation table is a table of references, relative to the start of the program. Each reference changes when the executable image is loaded into memory and an actual address for the entry point is assigned.)

Executable files linked with this option may be smaller, and thus load faster, than files linked without this option. Programs with many load-time relocations (about 500 or more) and long streams of repeated characters are usually shorter if packed. The /E option, however, does not always save a significant amount of disk space and sometimes may increase file size. LINK notifies you if the packed file is larger than the unpacked file.

Optimizing Far Calls (/F)

The /F option directs LINK to optimize far calls to procedures that lie in the same segment as the caller. Using the /F option may result in slightly faster code and smaller executable-file size. It should be used with the /PAC option for significant results. By default, the /F option is off.

For example, a medium- or large-model program may include a machine instruction that makes a far call to a procedure in the same segment. Because the instruction and the procedure it calls have the same segment address, only a near call is truly necessary. A near-call instruction does not require an entry in the relocation table as does a far-call instruction. In this situation, use of /F (together with /PAC) would result in a smaller executable file because the relocation table is smaller. Such files load faster.

When /F has been specified, LINK optimizes code by removing the following instruction:

```
call FAR label
```

LINK then substitutes the sequence:

```
nop
push cs
call NEAR label
```

Upon execution, the called procedure still returns with a far-return instruction. Because the code segment and the near address are on the stack, however, the far return is executed correctly. The `nop` (no-op) instruction appears so that exactly 5 bytes replace the 5-byte far-call instruction; LINK may in some cases place `nop` at the beginning of the sequence.

The /F option has no effect on programs that make only near calls. Of the high-level Microsoft languages, only small- and compact-model C programs use near calls.

Note

There is a small risk involved with the /F option: LINK may mistakenly translate a byte in a code segment that happens to have the far-call opcode (9A hexadecimal). If a program linked with /F inexplicably fails, then you may want to try linking with this option off. Object modules produced by Microsoft high-level languages, however, should be safe from this problem because relatively little immediate data is stored in code segments.

In general, assembly language programs are also relatively safe for use with the /F option, as long as they do not involve advanced system-level code, such as might be found in operating systems or interrupt handlers.

Viewing the Options List (/HE)

The /HE option causes LINK to display a list of its options on the screen. This gives you a convenient reminder of the options.

When you use this option, LINK ignores any other input you give and does not create an executable file.

Preparing for Incremental Linking (/INC)

The /INC option prepares a file for subsequent linking with ILINK. The use of this option produces a .SYM file and an .ILK file, each containing extra information needed by ILINK. Note that this option is not compatible with the /E option.

Displaying LINK Process Information (/INF)

The /INF option tells LINK to display information about the linking process, including the phase of linking and the names of the object files being linked. This option is useful if you want to determine the locations of the object files being linked and the order in which they are linked.

Output from this option is sent to the standard error output.

Example The following is a sample of LINK output when the /INF option is specified on the LINK command line:

```
**** PARSE DEFINITIONS FILE ****
**** PASS ONE ****
HELLO.OBJ(HELLO.OBJ)
**** LIBRARY SEARCH ****
BRT70ENR.LIB(..\rt\rtmdata.asm)
BRT70ENR.LIB(..\rt\rtmload.asm)
BRT70ENR.LIB(..\rt\rmessage.asm)
BRT70ENR.LIB(fixups.ASM)
BRT70ENR.LIB(..\rt\rtmint1.asm)
BRT70ENR.LIB(C:\TEMP\B6.)
**** ASSIGN ADDRESSES ****
**** PASS TWO ****
BRT70ENR.LIB(..\rt\rtmdata.asm)
BRT70ENR.LIB(..\rt\rtmload.asm)
BRT70ENR.LIB(..\rt\rmessage.asm)
BRT70ENR.LIB(fixups.ASM)
BRT70ENR.LIB(..\rt\rtmint1.asm)
BRT70ENR.LIB(C:\TEMP\B6.)
**** WRITING EXECUTABLE ****
Segments 36
Groups 1
Bytes in symbol table 10546
```

Including Line Numbers in the Map File (/LI)

You can include the line numbers and associated addresses of your source program in the map file by using the /LI option. This option is primarily useful if you will be debugging with the SYMDEB debugger included with earlier releases of Microsoft language products.

Ordinarily the map file does not contain line numbers. To produce a map file with line numbers, you must give LINK an object file (or files) with line-number information. (The /Zd and /Zi options of the compiler direct the compiler to include line numbers in the object file.) If you give LINK an object file without line-number information, the /LI option has no effect

The /LI option forces LINK to create a map file even if you did not explicitly tell LINK to create a map file. By default, the file is given the same base name as the executable file plus the extension .MAP. You can override the default name by specifying a new map file on the LINK command line or in response to the "List File" prompt.

Listing Public Symbols (/M)

You can list all public (global) symbols defined in the object file(s) by using the /M option. When you invoke LINK with the /M option, the map file contains a list of all the symbols sorted by name and a list of all the symbols sorted by address. LINK sorts the maximum number of symbols that can be sorted in available memory. If you do not use this option, the map file contains only a list of segments.

When you use this option, the default for the *mapfile* field or "List File" prompt response is no longer NUL. Instead, the default is a name that combines the base name of the executable file with a .MAP extension. You may still specify NUL in the *mapfile* field (which indicates that no map file is to be generated); if you do, the /M option has no effect.

Ignoring Default Libraries (/NOD:filename)

The /NOD option tells LINK *not* to search any library specified in the object file to resolve external references. If you specify *filename*, then LINK searches all libraries specified in the object file except for *filename*.

In general, higher-level language programs do not work correctly without a standard library. Therefore, if you use the /NOD option, you should explicitly specify the name of a standard library in the *libraries* field.

Ignoring Extended Dictionary (/NOE)

The /NOE option prevents LINK from searching the extended dictionary, which is an internal list of symbol locations that LINK maintains. Normally, LINK consults this list to speed up library searches. The effect of the /NOE option is to slow down LINK. You often need this option when a library symbol is redefined. Use /NOE if LINK issues the following error message:

```
symbol name multiply defined
```

Disabling Far-Call Optimization (/NOF)

This option is normally not necessary because far-call optimization (translation) is turned off by default. However, if an environment variable such as LINK turns on far-call translation automatically, you can use /NOF to turn far-call translation off again.

Preserving Case Sensitivity (/NOI)

By default, LINK treats uppercase letters and lowercase letters as equivalent. Thus, ABC, abc, and Abc are considered the same name. When you use the /NOI option, LINK distinguishes between uppercase letters and lowercase letters, and considers ABC, abc, and Abc to be three separate names. Because names in Microsoft BASIC are not case sensitive, this option can have minimal importance. You should not use the /NOI option when linking a protected-mode custom run-time module, or protected-mode program without the /O option.

Suppressing the Sign-On Logo (/NOL)

This option prevents the LINK sign-on banner from being displayed.

Ordering Segments Without Inserting Null Bytes (/NON)

This option directs LINK to arrange segments in the same order as they are arranged by the /DO option. The only difference is that the /DO option inserts 16 null bytes at the beginning of the _TEXT segment (if it is defined), whereas /NON does not insert these extra bytes.

Disabling Segment Packing (/NOP)

This option is normally not necessary because code-segment packing is turned off by default. However, if an environment variable such as LINK turns on code-segment packing automatically, you can use /NOP to turn segment packing off again.

Setting the Overlay Interrupt (/O:number)

By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The /O option allows you to select a different interrupt number.

The *number* can be a decimal number from 0 to 255, an octal number from octal 0 to octal 0377, or a hexadecimal number from hexadecimal 0 to hexadecimal FF. Numbers that conflict with DOS interrupts can be used; however, their use is not advised.

In general, you should not use /O with programs. The exception to this guideline would be a program that uses overlays and spawns another program that also uses overlays. In this case, each program should use a separate overlay-interrupt number, meaning that at least one of the programs should be compiled with /O.

Packing Contiguous Segments (/PAC:number)

The /PAC option affects code segments only in medium- and large-model programs. It is intended to be used with the /F option. It is not necessary to understand the details of the /PAC option in order to use it. You only need to know that this option, used in conjunction with /F, produces slightly faster and more compact code. The packing of code segments provides more opportunities for far-call optimization, which is enabled with /F. The /PAC option is off by default and can always be turned off with the /NOP option.

The /PAC option directs LINK to group neighboring code segments. Segments in the same group are assigned the same segment address; offset addresses are adjusted upward accordingly. In other words, all items have the correct physical address whether the /PAC option is used or not. However, /PAC changes segment and offset addresses so that all items in a group share the same segment address.

The *number* field specifies the maximum size of groups formed by /PAC. LINK stops adding segments to a group as soon as it cannot add another segment without exceeding *number*. At that point, LINK starts forming a new group. The default for *number* is 65,530.

Do not use this option if you have overlays. In addition, the /PAC option should not be used with assembly programs that make assumptions about the relative order of code segments. For example, the following assembly code attempts to calculate the distance between CSEG1 and CSEG2. This code would produce incorrect results when used with /PAC because /PAC causes the two segments to share the same segment address. Therefore, the procedure would always return 0.

```
CSEG1 SEGMENT PARA PUBLIC 'CODE'
  .
  .
  .
 CSEG1 ENDS

 CSEG2 SEGMENT PARA PUBLIC 'CODE'
 ASSUME cs:CSEG2

 ; Return the length of CSEG1 in AX.

 codsize PROC NEAR
 mov ax,CSEG2 ; Load para address of CSEG2
 sub ax,CSEG1 ; Load para address of CSEG1
 mov cx,4     ; Load count, and convert
 shl ax,cl    ; distance from paragraphs
              ; to bytes
 codsize ENDP

 CSEG2 ENDS
```

Packing Contiguous Data Segments (/PACKD)

This option only affects code segments in medium- and large-model programs and is safe with all Microsoft high-level language compilers. It behaves exactly like the /PAC option except that is applies to data segments, not code segments. LINK recognizes data segments as any segment definition with a class name that does not end in CODE. The adjacent data segment definitions are combined into the same physical segment up to the given limit. The default limit is 65,536.

Padding Code Segments (/PADC:padsize)

The /PADC option causes LINK to add filler bytes to the end of each code module for subsequent linking with ILINK. The option is followed by a colon and the number of bytes to add (a decimal radix is assumed, but you can specify octal or hexadecimal numbers by using a C-language prefix). Thus, the following adds an additional 256 bytes to each module:

```
/PADCODE:256
```

The default size for code-module padding is 0 bytes. To use this option, you must also specify the /INC option.

Note

Code padding is usually not necessary for large- and medium-model programs, but is recommended for small, compact, and mixed-memory model programs, and for assembly language programs in which code segments are grouped.

Padding Data Segments (/PADD:padsize)

The /PADD option performs a function similar to the /PADC option, except that it specifies padding for data segments (or data modules, if the program uses the small- or medium-memory model). The default size for data-segment padding is 16 bytes. To use the /PADD option, you must also specify the /INC option.

Note

If you specify too large a value for *padsize*, you may exceed the 64K limitation on the size of the default data segment.

Pausing During Linking (/PAU)

The /PAU option tells LINK to pause before it writes the executable file to disk. This option is useful on machines without hard disks, where you might want to create the executable file on a new disk. Without the /PAU option, LINK performs the linking session from beginning to end without stopping.

If you specify the /PAU option, LINK displays the following message before it creates the file:

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

The *letter* corresponds to the current drive. LINK resumes processing when you press Enter.

Note Do not remove the disk that will receive the listing file or the disk used for the temporary file.

Depending on how much memory is available, LINK may create a temporary disk file during processing, as described in the section "LINK Memory Requirements," later in this chapter and display the following message:

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter
```

If the file is created on the disk you plan to swap, press Ctrl+C to terminate the LINK session. Rearrange your files so that the temporary file and the executable file can be written to the same disk, then try linking again.

Specifying OS/2 Window Type (/PM:type)

The /PM option specifies the type of Presentation Manager window that the application can be run in. The argument *type* can be one of the following:

Type argument	Meaning
PM	Presentation Manager application. The application uses the Presentation Manager API and must be executed in the Presentation Manager environment. Not valid for BASIC programs.
VIO	Presentation Manager-compatible application. This application can run in the Presentation Manager environment from a VIO window, or it can be run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions supported in the Presentation Manager API. VIO applications written in BASIC are valid with the following restrictions: the application cannot support event handling or graphics. You can link your program with the NOEVENT.OBJ and NOGRAPH.OBJ stub files to remove these features.
NOVIO	Application is not compatible with the Presentation Manager and must operate in a separate screen group (default). Valid for BASIC programs. This option can be used in place of the WINDOAPI, WINDOWCOMPAT, and NOTWINDOWCOMPAT keywords in the module-definition file.

Creating a Quick Library (/Q)

When you use this option, LINK will create a Quick library that can be used from within the QBX environment. For instructions on how to create a Quick library, see Chapter 19, "Creating and Using Quick Libraries."

Setting Maximum Number of Segments (/SE:number)

The /SE option controls the number of segments that LINK allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1–3072 (decimal).

For each segment, LINK must allocate some space to keep track of segment information. By using a relatively low segment limit as a default (128), LINK is able to link faster and allocate less storage space.

When you set the segment limit higher than 128, LINK allocates additional space for segment information. This option allows you to raise the segment limit for programs with a large number of segments. For programs with fewer than 128 segments, you can keep the storage requirements of LINK at the lowest level possible by setting *number* to reflect the actual number of segments in the program. If the number of segments allocated is too high for the amount of memory available to LINK, LINK issues the following error message:

```
segment limit set too high
```

If this occurs, link the object files again, specifying a lower segment limit.

Issuing Fixup Warnings (/W)

This option directs LINK to issue a warning for each segment-relative fixup of location type *offset*, such that the segment is contained within a group but is not at the beginning of the group. LINK will include the displacement of the segment from the group in determining the final value of the fixup, unlike DOS executable files. Use this option when linking protected-mode programs or Windows applications.

To use this option with BASIC programs, you must also specify the /NOP option. This option should not be used to link custom run-time modules.

Object Files

The *objfiles* field allows you to specify the names of the object files you are linking. At least one object filename is required. A space or plus sign (+) must separate each pair of object filenames. LINK automatically supplies the .OBJ extension when you give a filename without an extension. If your object file has a different extension or if it appears in another directory or on another disk, you must give the full name—including the extension and path—for the file to be found. If LINK cannot find a given object file, and the drive associated with the object file is a removable-disk drive, then LINK displays a message and waits for you to change disks.

You may also specify one or more libraries in the *objfiles* field. To enter a library in this field, make sure that you include the .LIB extension; otherwise, LINK assumes the .OBJ extension. Libraries entered in this field are called "load libraries" as opposed to regular libraries. LINK automatically links every object module in a load library; it does not search for unresolved external references first. The effect of entering a load library is exactly the same as if you had entered the names of all the library's object modules in the *objfiles* field. This feature is useful if you are developing software using many modules and wish to avoid typing the name of each module on the LINK command line.

Executable File

The *exefile* field allows you to specify the name of the executable file. If the filename you give does not have an extension, LINK automatically adds .EXE as the extension (or .QLB if /Q is specified). You can give any filename you like; however, if you are specifying an extension, you should always use .EXE because the operating system expects executable files to have either this extension or the .COM extension.

Map File

The *mapfile* field allows you to specify the name of the map file if you are creating one. To include public symbols and their addresses in the map file, specify the /M option on the LINK command line.

If you specify a map filename without an extension, LINK automatically adds a .MAP extension. LINK creates the map file in the current working directory unless you specify a path for the map file.

Libraries

The *libraries* field allows you to specify the name of one or more libraries that you want linked with the object file(s). When LINK finds the name of a library in this field, it treats the library as a "regular library" and links only those object modules needed to resolve external references.

Each time you compile a source file for a high-level language, the compiler places the name of one or more libraries in the object file that it creates; LINK automatically searches for a library with this name (see the next section, "How LINK Searches for Libraries"). Because of this, you need not supply library names on the LINK command line unless you want to search libraries other than the default libraries or search for libraries in different locations.

When you link your program with a library, LINK pulls any library modules that your program references into your executable file. If the library modules have external references to other library modules, your program is linked with those other library modules as well.

How LINK Searches for Libraries

LINK searches for libraries that are specified in either of the following ways:

- In the *libraries* field on the command line or in response to the "Libraries" prompt.
- By an object module. BC writes the name of a default library in each object module it creates.

Note

The material in the following sections does not apply to libraries that LINK finds in the *objfiles* field, on the command line or in response to the "Object Modules" prompt. Those libraries are treated simply as a series of object files, and LINK does not conduct extensive searches in such cases.

Library Name with Path

If the library name includes a path, LINK searches only that directory for the library. Libraries specified by object modules (that is, default libraries) normally do not include a path.

Library Name Without Path

If the library name does not include a path, LINK searches the following locations, in the order shown, to find the library file:

1. The current directory.
2. Any paths or drive names that you give on the command line or type in response to the "Libraries" prompt, in the order in which they appear.
3. The locations given by the LIB environment variable.

Because object files created by BC contain the names of all the standard libraries you need, you are not required to specify a library on the LINK command line or in response to the LINK "Libraries" prompt unless you want to do one of the following:

- Add the names of additional libraries to be searched.
- Search for libraries in different locations.
- Override the use of one or more default libraries.

For example, if you have developed your own libraries, you might want to include one or more of them as additional libraries when you link them.

Searching Additional Libraries

You can tell LINK to search additional libraries by specifying one or more library files on the command line or in response to the "Libraries" prompt. LINK searches these libraries in the order you specify *before* it searches default libraries.

LINK automatically supplies the .LIB extension if you omit it from a library filename. If you want to link a library file that has a different extension, be sure to specify the extension.

For example, suppose that you want LINK to search the NEWLIBV3.LIB library before it searches the default library BCL70AFR.LIB. You would type LINK to start LINK, and then respond to the prompts as follows:

```
Object Modules .OBJ: SPELL TEXT DICT THES
Run File SPELL.EXE:
List File NUL.MAP:
Libraries .LIB: C:\TESTLIB\NEWLIBV3
```

This example links four object modules to create an executable filename SPELL.EXE. LINK searches NEWLIBV3.LIB before searching BCL70AFR.LIB to resolve references. To locate NEWLIBV3.LIB and the default libraries, LINK searches the current working directory, then the C:\TESTLIB\ directory, and finally the locations given by the LIB environment variable.

Searching Different Locations for Libraries

You can tell LINK to search additional locations for libraries by giving a drive name or path in the *libraries* field on the command line or in response to the "Libraries" prompt. You can specify up to 32 additional paths. If you give more than 32 paths, LINK ignores the additional paths without displaying an error message.

Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the name of a different library instead. You might need to specify a different library name in the following cases:

- You assigned a "custom" name to a standard library when you set up your libraries.
- You want to link with a library that supports a different math package than the math package you gave on the compiler command line (or the default).

If you specify a new library name on the LINK command line, LINK searches the new library to resolve external references before it searches the library specified in the object file.

If you want LINK to ignore the library whose name is included in the object file, you must use the /NOD option. This option tells LINK to ignore the default-library information that is encoded in the object files created by high-level language compilers. Use this option with caution; for more information, see the section "Ignoring Default Libraries (/NOD:filename)" earlier in this chapter.

Module-Definition File

The *deffile* field allows you to specify the name of a module-definition file (OS/2 protected-mode or Microsoft Windows programs only). Leave this field blank if you are linking a real mode program. The use of a module-definition file is optional for applications, but is required for dynamic-link libraries.

LINK Prompts

If you want LINK to prompt you for input, start LINK by typing the following at the system prompt:

```
LINK
```

LINK also displays prompts if you type an incomplete command line that does not end with a semicolon or if a response file (described in the section "LINK Response File" later in this chapter) is missing any required responses.

LINK prompts you for the input it needs by displaying the following lines, one at a time. The items in square brackets are the defaults LINK applies if you press Enter in response to the prompt. (You must supply at least one object filename for the "Object Modules" prompt.) LINK waits for you to respond to each prompt before it displays the next one.

```
Object Modules [.OBJ]:
Run File basename.[EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
Definitions File [NUL.DEF]:
```

Note that the default for the "Run File" prompt is the base name of the first object file with the .EXE extension.

The responses you give to the LINK command prompts correspond to the fields on the LINK command line as follows:

Prompt	Command-line field
Object Modules	*objfiles*
Run File	*exefile*
List File	*mapfile*
Libraries	*libraries*
Definitions File	*deffile*

If you type a plus sign (+) as the last character on a response line, the same prompt appears on the next line, and you can continue typing responses. The plus sign must appear at the end of a complete filename or library name, path, or drive name.

To select the default response to the current prompt, press the Enter key without giving a filename. The next prompt appears.

To select default responses to the current prompt and all remaining prompts, type a semicolon (;) and press Enter. After you type a semicolon, you cannot respond to any of the remaining prompts for that link session. This saves time when you want the default responses. Note, however, that you cannot enter only a semicolon in response to the "Object Modules" prompt because there is no default response for that prompt; LINK requires the name of at least one object file.

LINK Response File

A response file contains responses to the LINK prompts. The responses must be in the same order as the LINK prompts discussed in the previous section. Each new response must appear on a new line or must begin with a comma; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line. You may give options at the end of any response or place them on one or more separate lines.

LINK treats the input from the response file just as if you had entered it in response to prompts or on a command line. It treats any new-line character in the response file as if you had pressed Enter in response to a prompt or included a comma in a command line. For compatibility with OS/2 versions of LINK, it is recommended that all LINK response files end with a semicolon after the last line.

To use LINK with a response file, create the response file, then type the following command:

LINK @*responsefile*

Here *responsefile* specifies the name or path of the response file for LINK. You can also enter the name of a response file, preceded by an "at" sign (@), after any LINK command prompt or at any position in the LINK command line; in this case, the response file completes the remaining input.

Options and Command Characters

You can use options and command characters in the response file in the same way as you would use them in responses you type at the keyboard. For example, if you type a semicolon on the line of the response file corresponding to the "Run File" prompt, LINK uses the default responses for the executable file and for the remaining prompts.

Prompts

When you enter the LINK command with a response file, each LINK prompt is displayed on your screen with the corresponding response from your response file. If the response file does not include a line with a filename, semicolon, or carriage return for each prompt, LINK displays the appropriate prompt and waits for you to enter a response. When you type an acceptable response, LINK continues.

Example Assume that the following response file is named SPELL.LNK:

```
SPELL+TEXT+DICT+THES /PAUSE /MAP
SPELL
SPELLIST
XLIB.LIB;
```

You can type the following command to run LINK and tell it to use the responses in SPELL.LNK:

```
LINK @SPELL.LNK
```

The response file tells LINK to load the four object files SPELL, TEXT, DICT, and THES. LINK produces an executable file named SPELL.EXE and a map file named SPELLIST.MAP. The /PAU option tells LINK to pause before it produces the executable file so that you can swap disks, if necessary. The /M option tells LINK to include public symbols and addresses in the map file. LINK also links any needed routines from the library file XLIB.LIB. The semicolon is included after the library name for compatibility with the OS/2 version of LINK.

LINK Operation

LINK performs the following steps to combine object modules and produce an executable file:

1. Reads the object modules submitted.
2. Searches the given libraries, if necessary, to resolve external references.
3. Assigns addresses to segments.
4. Assigns addresses to public symbols.
5. Reads code and data in the segments.
6. Reads all relocation references in object modules.
7. Performs fixups.
8. Produces an executable file (executable image and relocation information).

Steps 5, 6, and 7 are performed concurrently: in other words, LINK moves back and forth between these steps before it progresses to step 8.

The "executable image" contains the code and data that constitute the executable file. The "relocation information" is a list of references, relative to the start of the program. The references change when the executable image is loaded into memory and an actual address for the entry point is assigned.

The following sections explain the process LINK uses to concatenate segments and resolve references to items in memory.

Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are BYTE, WORD, PARA, and PAGE. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively. The default alignment is PARA.

When LINK encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is WORD, PARA, or PAGE, LINK checks the executable image to see if the last byte copied ends on the appropriate boundary. If not, LINK pads the image with null bytes.

Frame Number

LINK computes a starting address for each segment in the program. The starting address is based on the segment's alignment and the sizes of the segments already copied to the executable file (as described in the previous section). The starting address consists of an offset and a canonical frame number. The "canonical frame number" specifies the address of the first paragraph in memory that contains one or more bytes of the segment. (A paragraph is 16 bytes of memory; therefore, to compute a physical location in memory, multiply the frame number by 16 and add the offset.) The offset is the number of bytes from the start of the paragraph to the first byte in the segment. For BYTE and WORD alignments, the offset may be nonzero. The offset is always zero for PARA and PAGE alignments. (An offset of zero means that the physical location is an exact multiple of 16.).

You can find the frame number for each segment in the map file created by LINK. The first four digits of the segment's start address give the frame number in hexadecimal. For example, a start address of 0C0A6 indicates the frame number 0C0A.

Order of Segments

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless LINK encounters two or more segments that have the same class name. Segments having identical segment names are copied as a contiguous block to the executable file.

The /DO option may change the way in which segments are ordered.

Combined Segments

LINK uses combine types to determine whether two or more segments that share the same segment name should be combined into one large segment. The valid combine types are PUBLIC, STACK, COMMON, and PRIVATE.

If a segment has combine type PUBLIC, LINK automatically combines it with any other segments that have the same name and belong to the same class. When LINK combines segments, it ensures that the segments are contiguous and that all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in the source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single, large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64K, LINK displays an error message.

If a segment has combine type STACK, LINK carries out the same combine operation as for PUBLIC segments. The only exception is that STACK segments cause LINK to copy an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has combine type COMMON, LINK automatically combines it with any other segments that have the same name and belong to the same class. When LINK combines COMMON segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment no larger than the largest segment combined.

A segment has combine type PRIVATE only if no explicit combine type is defined for it in the source file. LINK does not combine private segments.

Groups

Groups allow segments to be addressed relative to the same frame address. When LINK encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address.

Segments in a group do not have to be contiguous, belong to the same class, or have the same combine type. The only requirement is that all segments in the group fit within 64K.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee that the segments will be contiguous. In fact, LINK may place segments that do not belong to the group in the same 64K of memory. LINK does not explicitly check whether all the segments in a group fit within 64K of memory; however, LINK is likely to encounter a fixup-overflow error if they do not.

Fixups

Once LINK knows the starting address of each segment in the program and has established all segment combinations and groups, LINK can "fix up" any unresolved references to labels and variables. To fix up unresolved references, LINK computes the appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fixups for the types of references shown in Table 18.3.

Table 18.3 How LINK Fixes Unresolved References

Type of reference	Location of reference	LINK action
Short	In **JMP** instructions that attempt to pass control to labelled instructions in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference.	Computes a signed, 8-bit number for the reference, and displays an error message if the target instruction belongs to a different segment or group (has a different frame address), or if the target is more than 128 bytes away in either direction.
Near self-relative	In instructions that access data relative to the same segment or group.	Computes a 16-bit offset for the reference and displays an error if the data are not in the same segment or group.
Near segment-relative	In instructions that attempt to access data in a specified segment or group, or relative to a specified segment register.	Computes a 16-bit offset for the reference, and displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.
Long	In **CALL** instructions that attempt to access an instruction in another segment or group	Computes a 16-bit frame address and 16-bit offset for this reference, and displays an error message if the computed offset is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.

The size of the value to be computed depends on the type of reference. If LINK discovers an error in the anticipated size of a reference, it displays a fixup-overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction which is more than 64K away. It can also occur if all segments in a group do not fit within a single 64K block of memory.

LINK Memory Requirements

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, LINK creates a temporary disk file to serve as memory. This temporary file is handled in one of the following ways, depending on the DOS version

- For the purpose of creating a temporary file, LINK uses the directory specified by the TMP environment variable. If the TMP variable is set to C:\TEMPDIR, for example, then LINK puts the temporary file in C:\TEMPDIR.

  If there is no TMP environment variable or if the directory specified by TMP does not exist, then LINK puts the temporary file in the current directory.

- If LINK is running on DOS version 3.0 or later, it uses a DOS system call to create a temporary file with a unique name in the temporary-file directory.

- If LINK is running on a version of DOS prior to 3.0, it creates a temporary file named VM.TMP.

When LINK creates a temporary disk file, you see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter.
```

In the preceding message, *tempfile* is ".\" followed by either VM.TMP or a name generated by the system, and *letter* is the drive containing the temporary file.

If you are running on a floppy-disk system, the `Do not change diskette` message appears. After this message appears, do not remove the disk from the specified drive until the LINK session ends. If you remove the disk, the operation of LINK is unpredictable, and you may see the following message:

```
unexpected end-of-file on scratch file
```

If this happens, rerun the LINK operation. The temporary file created by LINK is a working file only. LINK deletes it at the end of the operation.

Note Do not give any of your own files the name VM.TMP. LINK displays an error message if it encounters an existing file with this name.

Linking Stub Files

Microsoft BASIC provides several special-purpose object files called "stub files" that you can use to minimize the size of your executable file in cases where your program does not use a particular BASIC feature or where special support is needed. By linking these files, you can make LINK exclude (or in some cases, include smaller versions of) code that it would otherwise place in your executable file automatically. Keep in mind that this process is appropriate only for programs compiled with the /O option.

You can also link stub files with custom run-time modules. Include the name of the file under the # OBJECTS directive in the export-file list for BUILDRTM. Table 18.4 lists the stub files included with Microsoft BASIC.

Table 18.4 Stub Files Included with BASIC

Filename	Description
OVLDOS21.OBJ	Adds DOS 2.1 support to an overlaid program.
NOCGA.OBJ	Removes support for a Color Graphics Adapter (CGA) features (screen modes 1 and 2).
NOCOM.OBJ	Removes communications support.
NOEDIT.OBJ	Removes the editor used with the INPUT and LINE INPUT statements.
NOEGA.OBJ	Removes Enhanced Graphics Adapter (EGA) support (screen modes 7, 8, 9, and 10).
NOEMS.OBJ	Prevents the overlay manager from using Expanded Memory Specification (EMS).
NOEVENT.OBJ	Removes event-handling support.
NOFLTIN.OBJ	Replaces the numeric parsing code with an integer-only version. If you link with NOFLTIN.OBJ, all numbers numbers used by **INPUT**, **READ**, and **VAL** must be legal long integers.
NOGRAPH.OBJ	Removes graphics features support.
NOHERC.OBJ	Removes Hercules graphics features (screen mode 3).
NOISAM.OBJ	Removes ISAM support.
NOLPT.OBJ	Removes line printer support.
NOOGA.OBJ	Removes Olivetti support.
NOVGA.OBJ	Removes Video Graphics Array (VGA) features (screen mode 11,12,13).
SMALLERR.OBJ	Reduces size of error messages displayed.
TSCNIO *string mode*	Removes certain features from BASIC programs to produce text-only screen I/O programs. This results in a smaller, simpler executable file. Four versions are provided, depending upon the far string support and mode selected: TSCNIONR Near string DOS TSCNIOFR Far string DOS TSCNIONP Near string protected mode TSCNIOFP Far string protected mode **Note:** These files cannot be used in conjunction with any of the graphics stub files.

Table 18.4 *Continued*

Filename	Description
87.LIB	Removes emulator code. Program linked with this file will run only on machines with a numeric coprocessor, and must be compiled with the /FPi option.
NOTRNEMR.LIB	Removes transcendental emulator support (real mode).
NOTRNEMP.LIB	Removes transcendental emulator support (protected mode).

For more information on stub files, see Chapter 15, "Optimizing Program Size and Speed."

Stub files (including the .LIB files listed) are specified in the *objfiles* field of LINK. You must supply the /NOE (/NOEXTDICTIONARY) option when linking any of the stub files. It is permissible to link more than one stub file at once. For instance, the following LINK command is appropriate for a program that requires no communications or printer support:

```
LINK /NOE NOLPT+NOCOM+MYPROG.OBJ,MYPROG.EXE;
```

This command links NOLPT.OBJ and NOCOM.OBJ to the user-created object file MYPROG.OBJ, producing the executable file MYPROG.EXE.

Linking with Overlays

You can direct LINK to create an overlaid version of a program. In an overlaid version of a program, specified parts of the program (known as "overlays") are loaded only if and when they are needed. These parts share the same space in memory. Only code is overlaid; data is never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to read and reread the code from disk into memory.

You specify overlays by enclosing them in parentheses in the list of object files that you submit to LINK. Each module in parentheses represents one overlay. For example, you could give the following object-file list in the *objfiles* field of the LINK command line:

```
A + (B+C) + (E+F) + G + (I)
```

In this example, the modules `(B+C)`, `(E+F)`, and `(I)` are overlays. The remaining modules, and any drawn from the run-time libraries, constitute the resident part (or root) of your program. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can appear only once in a program.

LINK replaces calls from the root to an overlay, and calls from an overlay to another overlay, with an interrupt (followed by the module identifier and offset). By default, the interrupt number is 63 (3F hexadecimal). You can use the /O option of the LINK command to change the interrupt number.

The CodeView debugger is compatible with overlaid modules. In fact, in the case of large programs, you may need to use overlays to leave sufficient room for the debugger to operate. When you link overlaid code using the /CO option, you will receive an error message `Multiple code segments in module of overlaid code`. This is normal.

Care should be taken to compile each module in the program with compatible options. This means, for example, that all modules must be compiled with the same floating-point options.

Using Expanded Memory

If expanded memory is present in your computer, overlays are loaded from expanded memory; otherwise, overlays are loaded from disk. You can specify that overlays only be loaded from disk by linking your program with the NOEMS.OBJ stub file.

If your program uses overlays from Expanded Memory Specification (EMS), and if it contains a routine that changes the state of EMS (for example, an assembly language routine that shells out to another program), you must restore the state of EMS before returning to the overlaid code. To do this, call the **B_OVREMAP** routine. This routine restores EMS to the state that existed before the routine that changed the state was called, and insures that overlays are loaded from EMS correctly. **B_OVREMAP** has no effect if overlays are not used or if overlays are not loaded from EMS.

Restrictions on Overlays

The following restrictions apply to using overlays in Microsoft BASIC:

- Each Microsoft BASIC overlay cannot be larger than 256K. There is a maximum of 64 overlays per program.
- Overlays should not be specified as the first object module on the LINK command line (the first object module must be a part of the program that is not overlaid).
- When you create an overlaid version of a program, make sure that each module contained in the program is compiled with compatible options.
- You cannot use the /PACKCODE option when linking a program that uses overlays.
- You can overlay only modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. Also, LINK does not produce overlay modules that can be called indirectly through function pointers. When a function is called through a pointer, the called function must be in the same overlay or root.

Overlay-Manager Prompts

The overlay manager is part of the language's run-time library. If you specify overlays during linking, the code for the overlay manager is automatically linked with the other modules of your program. Even with overlays, LINK produces only one .EXE file. At run time, the overlay manager opens the .EXE file each time it needs to extract new overlay modules. The overlay manager first searches for the file in the current directory; then, if it does not find the file, the manager searches the directories listed in the PATH environment variable. When it finds the file, the overlay manager extracts the overlay modules specified by the root program. If the overlay manager cannot find an overlay file when needed, it prompts you for the filename.

For example, assume that an executable program named PAYROLL.EXE uses overlays and does not exist in either the current directory or the directories specified by PATH. If your program does not contain expanded memory, when you run PAYROLL.EXE (by entering a complete path), the overlay manager displays the following message when it attempts to load overlay files:

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

You can then enter the drive or directory, or both, where PAYROLL.EXE is located. For example, if the file is located in directory \EMPLOYEE\DATA\ on drive B, you could enter `B:\EMPLOYEE\DATA\` or simply `\EMPLOYEE\DATA\` if the current drive is B.

If you later remove the disk in drive B and the overlay manager needs to access the overlay again, it does not find PAYROLL.EXE and displays the following message:

```
Please insert diskette containing B:\EMPLOYEE\DATA\PAYROLL.EXE
and strike any key when ready.
```

After reading the overlay file from the disk, the overlay manager displays the following message:

```
Please restore the original diskette.
Strike any key when ready.
```

Execution of the program then continues.

Chapter 19
Creating and Using Quick Libraries

This chapter describes how to create and use Quick libraries. You'll learn how to do the following:

- Make libraries from within the QBX environment and from the command line.
- Make a Quick library that contains routines from an existing Quick library.
- Load a Quick library when running a QBX program.
- View the contents of a Quick library.

Also, specific examples of how to create Quick libraries from different types of source code modules are presented and the last section of this chapter provides programming information specific to Quick libraries.

For an overview of Quick libraries and reasons why you might want to use them, see Chapter 17, "About Linking and Libraries."

The Supplied Library (QBX.QLB)

Microsoft BASIC supplies a default Quick library named QBX.QLB. If you invoke QBX with the /L option, but do not supply a Quick library name, QBX automatically loads the library QBX.QLB, included with the QBX package. This file contains three routines, **INTERRUPT**, **INT86OLD**, and **ABSOLUTE**, that provide software-interrupt support for system-service calls and support for **CALL ABSOLUTE**. QBX.QLB is also necessary for creating certain other Quick libraries (see the section "Mouse, Menu, and Window Libraries" later in this chapter).

You must load QBX.QLB (or another library into which **INTERRUPT**, **INT86OLD**, and **ABSOLUTE** have been incorporated) from the command line when you invoke QBX in order to use the routines from QBX.QLB. If you wish to use these routines along with other routines that you have placed in libraries, make a copy of the QBX.QLB library and use it as a basis for building a library containing all the routines you need.

A parallel object-module library, QBX.LIB, is also supplied for use outside of the QBX environment.

Files Needed to Create a Quick Library

To create a Quick library, make sure that the following files are in the current working directory or accessible to QBX through the appropriate DOS environment variables:

File	Purpose
QBX.EXE	Directs the process of creating a Quick library. If you are working only with QBX modules, you can do everything in one step from within the QBX environment.
BC.EXE	Creates object files from source code.
LINK.EXE	Links object files.
LIB.EXE	Manages object-module libraries of object modules.
QBXQLB.LIB	Supplies routines needed by your Quick library. This library is a object-module library that is linked with objects in your library to form a Quick library.

In addition, if you are creating a Quick library from QBX and use the /L option to load a Quick library, you must also make sure that the parallel object-module library is also accessible to QBX.

If you are creating Quick libraries from sample source file modules, read "Mouse, Menu, and Window Libraries" later in this chapter for information about additional files you'll need to have.

Types of Source Files

You can create a Quick library from three types of source files:

- BASIC source files (.BAS).
- Object files (.OBJ). The source files for these can be written in any Microsoft language, such as BASIC or C, before being compiled with the appropriate compiler.
- Library (.LIB) files.

If you have only BASIC source files, you can create the Quick library from either QBX or from the command line. However, if you have object files or libraries, you must perform this task from the command line.

Note

Routines to be placed in Quick libraries must be compiled with the following options: /FPi, /Fs (if strings are used), and /Lr (the default). Routines compiled with the /FPa and /Lp options of BASIC Compiler (BC) cannot be placed into Quick libraries.

The .QLB Filename Extension

The extension .QLB is just a convenient convention. You can use any extension for your Quick library files, or no extension at all. However, in processing the /L *libraryname* option, QBX assumes that the listed *libraryname* has the .QLB extension if no other extension is specified. If your Quick library has no extension, you must put a period after the Quick library name (*libraryname.*) or QBX searches for a file with your base name and the .QLB extension.

Creating a Quick Library

A Quick library can be created from within QBX or by invoking command line utilities. The sections that follow describe the basic steps for each and include the examples of how to combine various types of source files into a Quick library.

Only whole modules can be put into a Quick library. That is, you cannot select one procedure from among many in a module. If you want to enter only certain procedures from a module, put the procedures you want in a separate module, then put that module into a library.

A Quick library must be self-contained. A procedure in a Quick library can only call other procedures within the same Quick library. Procedure names must be unique within the library. You can, however, combine the routines contained in several libraries.

With large programs, you can reduce loading time by putting as many routines as possible into Quick libraries. This is also an advantage if you want the program to be a stand-alone executable file later, since the contents of libraries are simply linked without recompiling.

Note

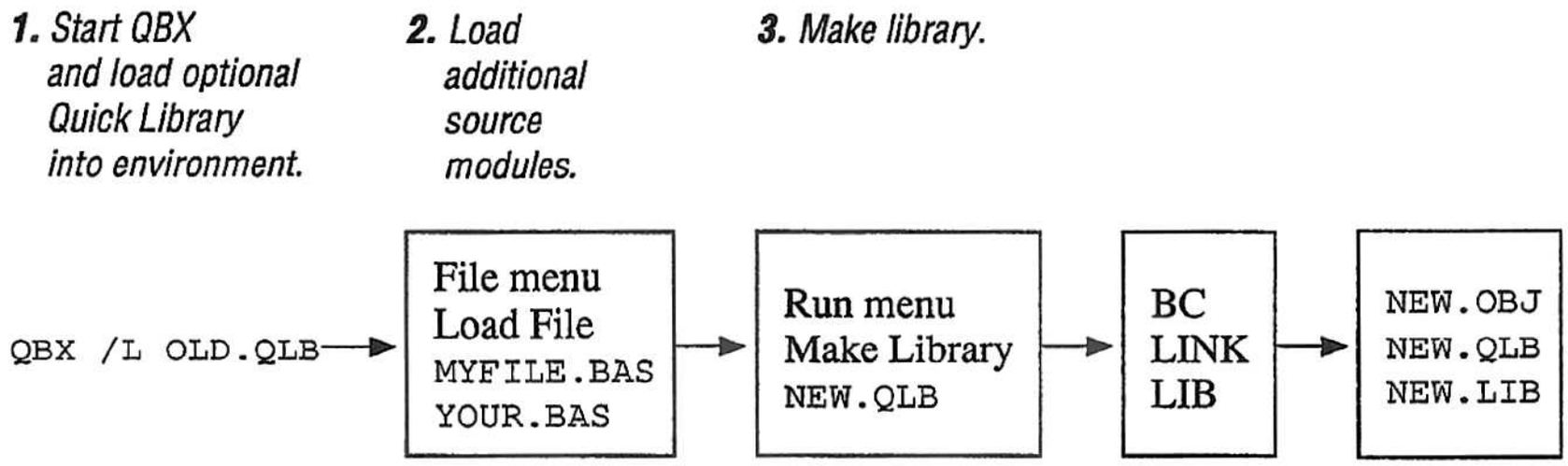
Your main module may or may not contain procedures. If it does and you incorporate those procedures into the library, the entire main module goes in the library, too. This does not cause an error message, but the module-level code in the library can never be executed unless one of its procedures contains a routine (such as **ON ERROR**) that explicitly passes control to the module level. Even if that is the case, much of the module-level code may be extraneous. If you organize your procedures in modules that are frequently used together, your Quick libraries are likely to be less cluttered with useless code.

Making a Quick Library from QBX

A Quick library automatically contains all modules, the main module and subordinate modules, present in the QBX environment when you create the library. It also contains the contents of any other Quick library that you loaded when starting QBX. If you load a whole program but only want certain modules to be put in the library, you must explicitly unload those you don't want. You can unload modules with the File Unload File command.

You can quickly determine which modules are loaded by selecting the list box of the SUBs command on the View menu. However, this method does not show which procedures a loaded library contains. The QLBDUMP.BAS utility program, described in the section "Viewing the Contents of a Quick Library" later in this chapter, allows you to list all the procedures in a library.

The basic steps to creating a Quick library from QBX are as follows:



Starting QBX

When making a library from within the QBX environment, the first consideration is whether the library will be composed of new modules or if you want to place routines from an existing Quick library into a new Quick library.

If you are creating a Quick library from new modules, start QBX without the /L command-line option.

If you want to include routines from an existing Quick library, you should start QBX with the /L command-line option, supplying the name of the existing Quick library as a command-line argument.

You can optionally include the name of a program whose modules you want to put in the library. In this case QBX loads all the modules specified in that program's .MAK file.

Loading Desired Files

The easiest way to create a Quick library is to start QBX, with or without a Quick library (/L) specification and load the modules you want one at a time from within the environment. In this case, you load each module using the Load File command from the File menu.

To load one module at a time with QBX:

1. From the File menu, choose Load File.
2. Select the name of a module you want to load from the list box.
3. Repeat steps 1 and 2 until all you have loaded all the modules you want.

Unloading Unwanted Files

Alternately, you can load your program when starting QBX, and unload any modules you don't want in the Quick library, including the main module (unless it contains procedures you want in the library).

Follow these steps to unload modules:

1. From the File menu, choose Unload File.
2. Select the module you want to unload from the list box, then press Enter.
3. Repeat steps 1 and 2 until you have unloaded all unwanted modules.

Creating a Quick Library

Once you have loaded the previous library (if any) and all the new modules you want to include in the Quick library, choose the Make Library command from the Run menu. Then, perform the following steps:

1. Enter the name of the library you wish to create in the Library File Name text box.

   If you enter only a base name (that is, a filename with no extension), QBX automatically appends the extension .QLB when it creates the library. If you want your library to have no extension, add a terminating period (.) to the base name. Otherwise, you may enter any base name and extension you like (except the name of a loaded Quick library), consistent with DOS file naming rules.
2. Select any options you wish from the Arrays and Strings or Other/Debug text boxes. Press F1 for help information about the options listed.
3. Create the Quick library. Choose the Make Library command button if you want to remain in the environment after the Quick library is created. Choose the Make Library and Exit command button if you want to return to the DOS command level after the Quick library is created.

Note When you make a Quick library, be aware that if it ever will be used with a program module that needs to trap events such as keystrokes, then one of the modules in the library must contain at least one event-trapping statement. This statement can be as simple as **TIMER OFF**, but without it, events are not trapped correctly in the Quick library.

Additional Files Created

After making a library from within the QBX environment, you will notice the appearance of extra files with the extensions .OBJ and .LIB. In creating Quick libraries, QBX actually directs the work of three other programs: BC, LINK, and LIB, and then combines what they produce into a Quick library and an object-module library. Once the process is complete, there is one object file (.OBJ) for each module in your Quick library and a single object-module library file (.LIB) containing an object module for each object file. The files with the extension .OBJ are now extraneous and can be deleted. However, files with the extension .LIB are very important and should be preserved. These parallel libraries are the files QBX uses to create executable files of your programs.

Note Professional software developers should be sure to deliver both the Quick (.QLB) and object-module (.LIB) versions of libraries to customers. Without the object-module libraries, users would not be able to use the Quick library routines in executable files produced with QBX.

Making a Quick Library from the Command Line

You can use the LINK and LIB utilities to create both Quick libraries and object-module libraries from the command line. You will need to do this if you have modules containing routines written in other Microsoft languages, such as C or Macro Assembler (MASM), or if you are making a Quick library from an add-on library.

The basic steps for creating a Quick library (and the parallel object-module library) from the command line are as follows:

1. Compile or assemble all source files you wish to include in the Quick library.
2. Invoke the LINK utility to create the Quick library. List the names of the object modules you want in the Quick library. Use the /Q option to direct LINK to produce a Quick library. You must specify the QBXQLB.LIB library after the third comma on the LINK command line or in response to the Libraries prompt.
3. Invoke the LIB utility to create a parallel object-module library (for creating stand-alone executable files).

Building a New Quick Library

This section describes several ways that you can build Quick libraries from QBX and from the command line. Several examples illustrate how to build Quick libraries from BASIC toolbox files and libraries supplied with Microsoft BASIC. For specific information about the LINK and LIB utilities, see Chapter 18, "Using LINK and LIB."

Making a New Quick Library from BASIC Source Modules (QBX)

You can make a Quick library from QBX if you have only BASIC source modules that you wish to place in the library. For example, the file MOUSE.BAS is supplied with Microsoft BASIC. You can build a Quick library of mouse routines from MOUSE.BAS by doing the following:

1. Start QBX, specifying MOUSE.BAS as the filename and use the /L option to load the QBX.QLB Quick library:

   ```
   QBX /L QBX.QLB
   ```

   The mouse library requires the routines found in QBX.QLB (see "Mouse, Menu, and Window Libraries" later in this chapter for more information).

2. From the File menu, choose Load File. Load MOUSE.BAS into the environment.

3. From the Run menu, choose Make Library and enter the name of the new Quick library in the Library File Name field. Select any options you like from the Arrays and Strings or Other/Debug list boxes (see online Help for information about these options).

4. Choose Make Library (or Make Library and Exit) to start building the Quick library.

Notice that QBX invokes BC, LINK, and LIB to compile the BASIC source files, create the Quick library and the object-module library.

Making a New Quick Library from BASIC Source Modules (Command Line)

You can also create a Quick library by invoking the BC, LINK, and LIB utilities separately on the command line. To create the mouse library shown in the previous example, perform the following steps:

1. Compile the MOUSE.BAS file using BC:

   ```
   BC MOUSE.BAS /Fs;
   ```

2. Create the Quick library MOUSE.QLB by invoking LINK:

   ```
   LINK /Q MOUSE.OBJ, MOUSE.QLB, ,QBX.LIB QBXQLB.LIB;
   ```

Note that the object-module library QBX.LIB is included in the *libraries* field of LINK. When you place QBX.LIB in the *libraries* field of LINK, then only those QBX.LIB procedures that are referenced by MOUSE.BAS are placed in the Quick library. Thus, when MOUSE.QLB is loaded into the QBX environment, your interpreted code cannot reference QBX.LIB procedures that have not been pulled in. To reference those procedures (that is, QBX.LIB procedures not referenced by MOUSE.BAS), you must specify QBX.LIB in the *objects* field of LINK as follows:

```
LINK /Q MOUSE.OBJ QBX.LIB, MOUSE.QLB,,QBXQLB.LIB;
```

Specifying QBX.LIB in the *libraries* field of LINK results in a smaller Quick library, because unnecessary procedures are not pulled into MOUSE.QLB.

3. Create the object-module library MOUSE.LIB using LIB:

```
LIB MOUSE.LIB+MOUSE.OBJ+QBX.LIB;
```

Making a Quick Library from Other-Language Modules (Command Line)

To place routines from other languages in a Quick library, you must start with compiled or assembled object files that contain the routines you wish to use. Several other languages are suitable for this purpose, including Microsoft C, MASM, Pascal, FORTRAN, and any other language that creates object files compatible with the Microsoft language family.

Microsoft BASIC supplies an assembly language object file, UIASM.OBJ, which is needed to build the General, Menu, and Window source module libraries. To place this object file into a Quick library, you must do the following:

1. Link the object file and specify the /Q option, to produce a Quick library file:

   ```
   LINK /Q UIASM.OBJ QBX.LIB, MIXED.QLB,,QBXQLB.LIB;
   ```

   LINK interprets the entry that follows the name of the object file (in this case MIXED.QLB) as the filename by which the linked modules will be known. In this case, the Quick library file will be named MIXED.QLB. Note that the QBX library is specified in the object file field of LINK.

2. Now create a parallel object-module library, using the same object file you used to make the Quick library. In this case, the first name following the LIB command (MIXED.LIB) will be the name of the newly created object-module library.

   ```
   LIB MIXED.LIB+UIASM.OBJ+QBX.LIB;
   ```

   It is easy to overlook this step when making a library that contains other-language routines, but this step is crucial if you hope to use the library to create a stand-alone executable file. Without these parallel object-module libraries, QBX cannot create an executable file containing their routines.

See the section "Handling Strings and Arrays" later in this chapter for special considerations when writing other-language routines intended for use as Quick libraries.

Making a Quick Library from BASIC and Other-Language Modules (QBX)

The Menu library consists of routines in the assembly object module UIASM.OBJ, as well as routines in the QBX.LIB library and in the BASIC source files MENU.BAS, MOUSE.BAS, and GENERAL.BAS. To make these files into a Quick library, you must do the following:

1. Link UIASM.OBJ and QBX.QLB to create a Quick library, as shown in the previous example.
2. Start QBX and load this Quick library into the environment using the /L option of QBX as follows:

   ```
   QBX /L MIXED.QLB
   ```

3. Load MENU.BAS, MOUSE.BAS, and GENERAL.BAS into the environment and make the library, following the steps listed in the section "Making a Quick Library from QBX" found earlier in this chapter.

See the section "Handling Strings and Arrays" later in this chapter for special considerations when writing other-language routines intended for use as Quick libraries.

Making a Quick Library from .LIB Files (Command Line)

You can create Quick libraries from object-module library files as well as from object files. For example, Microsoft BASIC supplies the library DTFMTER.LIB, which contains date and time functions, and the FINANCER.LIB library, which contains financial functions (several versions are supplied to support various floating-point and mode options).

To create a Quick library from the DTFMTER.LIB library, invoke LINK as follows:

```
LINK /Q DTFMTER.LIB,DTFMTER.QLB,,QBXQLB.LIB;
```

This creates the Quick library DTFMTER.QLB.

If you later want to add the routines in the financial library to the date/time library, you must link both libraries into a single Quick library as follows:

```
LINK /Q DTFMTER.LIB FINANCER.LIB, DT_FIN.QLB,,QBXQLB.LIB;
```

This creates the combined Quick library DT_FIN.QLB. You cannot directly add the contents of an object-module library to an existing Quick library. You must create a new Quick library from the two object-module libraries.

Making a Quick Library from Other Quick Libraries (Command Line)

You may want to place routines from existing Quick libraries into a new Quick library. Since objects in Quick libraries cannot be manipulated, you must work with a Quick library's parallel object-module library.

For example, say that you have a Quick library called SAMPLE.QLB. You want to extract the modules BIG and BAR and place them into a new Quick library To do this, you need to do the following:

1. Copy the modules FIG and BAR from the object-module library SAMPLE.LIB to object files as follows:

   ```
   LIB SAMPLE *FIG *BAR;
   ```

   LIB places `FIG` into the file FIG.OBJ and `BAR` into the file BAR.OBJ.

2. Then, link the object modules as follows:

   ```
   LINK /Q FIG.OBJ BAR.OBJ, NEW.QLB,,QBXQLB.LIB;
   ```

The new Quick library is called NEW.QLB. You should also make a parallel object-module library (NEW.LIB) as shown in previous examples.

Combining BASIC Source Modules and Existing Quick Library Modules (QBX)

You can combine routines in a Quick library with BASIC source modules to create a new Quick library. For example, you might want to create a windowing library that contains the menu functions contained in a menu library.

For example, suppose that you already have built a Quick library named MENU.QLB. You wish to combine the routines found in the BASIC toolbox file WINDOW.BAS with the routines in MENU.QLB to create a new Quick library. You would do the following:

1. Load the menu Quick library into QBX:

   ```
   QBX /L MENU.QLB
   ```

2. Load the WINDOW.BAS file into the environment and make the library, as described in the section "Making a New Quick Library from BASIC Source Modules (QBX)" earlier in this chapter. Your new Quick library must have a name that is different from the Quick library that is currently loaded. Otherwise, QBX will reject the name and make you choose another name.

Mouse, Menu, and Window Libraries

If you planning on building mouse, menu, or window routine libraries from the BASIC toolbox files supplied with Microsoft BASIC, keep in mind that you must link them with other source modules to create a Quick library.

Table 19.1 lists additional files that must be linked with MOUSE.BAS, MENU.BAS, or WINDOW.BAS to create a mouse, menu, or window routine Quick library, and includes files that must be inserted in programs that will use the library:

Table 19.1 Requirements for Using BASIC Toolbox Files

If you use:	You must link with:	And you must include:
MOUSE.BAS	QBX.QLB	MOUSE.BI
MENU.BAS	MOUSE.BAS, GENERAL.BAS, UIASM.OBJ, QBX.QLB	MENU.BI, MOUSE.BI, GENERAL.BI
WINDOW.BAS	MENU.BAS, MOUSE.BAS, GENERAL.BAS, UIASM.OBJ, QBX.QLB	WINDOW.BI, MENU.BI, MOUSE.BI, GENERAL.BI
CHRTB.BAS	CHRTBEFR.LIB	CHRTB.BI
FONTB.BAS	FONTEFR.LIB	FONTB.BI

Loading and Viewing Quick Libraries

This section explains how to load a Quick library when you start QBX and how to view the contents of a Quick library. It also gives facts that you should remember when procedures within a Quick library perform floating-point arithmetic.

Loading a Quick Library

To load a Quick library, you must specify the name of the desired library on the command line when you start QBX using the following syntax:

QBX [[*programname*]] /L [[*quicklibraryname*]] [[/RUN]]

If you start QBX with the /L option and supply the name of a Quick library (*quicklibraryname*), QBX loads the specified Quick library and places you in the programming environment. The contents of the library are now available for use. If you start QBX with the /L option but don't specify a library, QBX loads the default library QBX.QLB (see the section "The Supplied Library (QBX.QLB)" earlier in this chapter).

Example To use the Quick library MOUSE.QLB in a BASIC program, type the name of library after the /L option of QBX. For example, if you wanted to start QBX, edit the file NEW.BAS, and load the library MOUSE.QLB, you would type the following:

```
QBX NEW.BAS /L MOUSE.QLB
```

Any routines found in MOUSE.QLB would now be available to use in your program.

Using the /RUN Option

You can also start QBX with the /RUN option followed by both a program name (*programname*) and the /L option. In this case, QBX loads the program and the specified Quick library, then runs the program before stopping in the programming environment.

Note

When using Quick libraries to represent program modules, remember to update the .MAK file to keep it consistent with the modules in the evolving program. (This is done with the Unload File command from the File menu.) If the .MAK file is not up to date, it may cause QBX to load a module containing a procedure definition with the same name as one defined in the Quick library, which in turn causes the error message `Duplicate definition`.

Quick Library Search Path

You can load only one Quick library at a time. If you specify a path, QBX looks there first; otherwise, QBX searches for the Quick library in the following three locations:

1. The current directory.
2. The path specified for libraries by the Set Paths command.
3. The path specified by the LIB environment variable. (See your operating system documentation for information about environment variables.)

Viewing the Contents of a Quick Library

Because a Quick library is essentially a binary file, you cannot view its contents with a text editor to find out what it contains. One of your distribution disks includes the QLBDUMP.BAS utility, which allows you to list all the procedures and data symbols in a given library. Follow these steps to view the contents of a Quick library:

1. Start QBX.
2. Load and run QLBDUMP.BAS.
3. When QBX prompts you, enter the name of the Quick library you wish to examine. You do not need to include the .QLB extension when you type the filename; however, supplying the extension does no harm.

   If the specified file exists and it is a Quick library, the program displays a list of all the symbol names in the library. In this context, symbol names correspond to the names of procedures in the library.

See Chapter 3,"File and Device I/O," for a commented listing of QLBDUMP.BAS.

Programming Considerations for Quick Libraries

You can write your Quick library in Microsoft BASIC or other Microsoft languages. This section provides additional information you might find helpful while developing your Quick library.

The B_OnExit Routine

QBX provides a BASIC system-level function, the **B_OnExit** routine. You can use **B_OnExit** when your other-language routines take special actions that need to be undone before leaving the program (intentionally or otherwise) or rerunning the program. For example, within the QBX environment, an executing program that calls other-language routines in a Quick library may not always run to normal termination. If such routines need to take special actions at termination (for example, the removal of previously installed interrupt vectors), you can guarantee that your termination routines will always be called if you include an invocation of **B_OnExit** in the routine.

Examples

The following example illustrates such a call (for simplicity, the example omits error-handling code). Note that such a function would be compiled in Microsoft C in large model.

```
#include <malloc.h>
extern pascal far B_OnExit(); /* Declare the routine*/
int *p_IntArray;
void InitProc()
{
   void TermProc(); /* Declare TermProc function  */
   /* Allocate far space for 20-integer array: */
   p_IntArray = (int *)malloc(20*sizeof(int));
   /* Log termination routine (TermProc) with BASIC: */
   B_OnExit(TermProc);
}
                        /* The TermProc function is*/
void TermProc()         /* called before any restarting*/
{                       /* or termination of program.*/
   free(p_IntArray);    /* Release far space allocated*/
}                       /* previously by InitProc.*/
```

If the `InitProc` function were in a Quick library, the call to **B_OnExit** would insure proper release of the space reserved in the call to **malloc**, should the program crash. The routine could be called several times, since the program can be executed several times from the QBX environment. However, the `TermProc` function itself would be called only once each time the program runs.

The following BASIC program is an example of a call to the `InitProc` function:

```
DECLARE SUB InitProc CDECL

X = SETMEM(-2048)    ' Make room for the malloc memory
                     ' allocation in C function.
CALL InitProc
END
```

If more than 32 routines are registered, **B_OnExit** returns NULL, indicating there is not enough space to register the current routine. (Note that **B_OnExit** has the same return values as the Microsoft C run-time library routine **ONEXIT**.)

B_OnExit can be used with any other-language (including assembly language) routines you place in a Quick library. With programs compiled and linked completely from the command line, **B_OnExit** is optional.

Handling Strings and Arrays

Other-language routines that are going to be used as Quick libraries must use the BASIC far string calls if they exchange or manipulate BASIC strings. For more information on using far strings calls, see Chapter 13. "Mixed-Language Programming with Far Strings."

If you are using a Quick library with containing an other-language routine that receives an array created in BASIC, you cannot invoke QBX with the /Ea (arrays in EMS) option.

Determining the Maximum Size for a Quick Library

Because a Quick library is essentially an executable file (although it cannot be invoked by itself from the DOS command line), it is quite large in comparison to the sum of the sizes of its source files. This puts an upper limit on the number of routines you can put in a Quick library. To determine how large your Quick library can be, add up the memory required for DOS, QBX.EXE, and your program's main module. An easy way to estimate these factors is to boot your machine, start QBX with your program, and enter this command in the Immediate window:

```
PRINT FRE (-1)
```

This command shows you the number of bytes of free memory. This indicates the maximum size for any Quick library associated with this program. In most cases, the amount of memory required for a Quick library is about the same as the size of its disk file. One exception to this rule of thumb is a library with procedures that use a lot of strings; such a program may require somewhat more memory.

Making Compact Executable Files

When QBX creates a Quick library, it also creates an object-module library of object modules in which each object module corresponds to one of the modules in the Quick library. When you make an executable file, QBX searches the object-module library for object modules containing the procedures used in the program.

If an object module in the library does not contain procedures referred to in the program, it is not included in the executable file. However, a single module may contain many procedures, and if even one of them is used in the program all are included in the executable file. Therefore, even if a program uses only one of four procedures in a certain module, that entire module becomes part of the final executable file.

To make your executable files as compact as possible, you should maintain a library in which each module contains only closely related procedures. If you have any doubts about what a library contains, list its contents with the utility QLBDUMP.BAS, described in the section "Viewing the Contents of a Quick Library" earlier in this chapter.

Chapter 20
Using NMAKE

The Microsoft Program Maintenance Utility (NMAKE) is an intelligent command processor that can save you time and simplify project management. By comparing the dates and times of files that you specify, NMAKE determines which project files to rebuild. Then it updates your application by executing commands that you specify.

The advantage of NMAKE over simple batch files is that NMAKE does only what is needed. You don't waste time rebuilding files that are already up to date. NMAKE also has advanced features, such as macros, that help you manage complex projects.

If you are familiar with the Microsoft MAKE utility, the predecessor of NMAKE, be sure to read "Differences Between NMAKE and MAKE," later in this chapter; there are some important differences between the two utilities.

NMAKE in a Nutshell

NMAKE works by comparing the times and dates of two sets of files, which are called "targets" and "dependents." A target is a file that you want to create, such as an executable file. A dependent is a file used to create a target, such as a BASIC source file.

When you run NMAKE, it reads a "description file" that you supply. The description file consists of one or more blocks. Each block typically lists a target, the target-dependents, and the command that builds the target. NMAKE compares the date and time of the target to those of its dependents. If any dependent has changed more recently than the target, NMAKE updates the target by executing the command listed in the block.

NMAKE's main purpose is to help you update applications quickly and simply. However, it can execute any command, so it is not limited to compiling and linking. NMAKE can also make backups, move files, and do many other project management tasks.

You invoke NMAKE from the DOS or OS/2 command line, specifying any options, macro definitions, and names of targets you wish to build on the command line. If you prefer, you can create a file containing options, macro definitions, and names of targets, and specify the name of this file on the NMAKE command line (see the section "Invoking NMAKE Using a File" later in this chapter for more information on how to do this).

Invoking NMAKE from the Command Line

The syntax for invoking NMAKE from the command line is as follows:

NMAKE [[*options*]] [[*macrodefinitions*]] [[*target...*]] [[*filename*]]

Argument	Description
options	An optional field that specifies options that modify the action of NMAKE. They are described under "NMAKE Options" later in this chapter.
macrodefinitions	An optional field that lists macro definitions for NMAKE to use. Macros provide a convenient method for replacing a string of text in the description file. Macro definitions that contain spaces must be enclosed by quotation marks. Macros are discussed in "Macros" later in this chapter.
target...	An optional field that specifies the name of one or more targets to build. If you do not list any targets, NMAKE builds the first target in the description file.
filename	An optional field that gives the name of the description file from which NMAKE reads target- and dependent-file specifications and commands. A better way of designating the description file is to use the /F option (see "NMAKE Options" for more information about this option). By default, NMAKE looks for a file named MAKEFILE in the current directory. If MAKEFILE does not exist, NMAKE uses the *filename* field: it interprets the first string on the command line that is not an option or macro definition as the name of the description file, provided its filename extension isn't listed in the .SUFFIXES list. (See the section "Pseudotargets" later in this chapter for more information about the .SUFFIXES list.)

Note Unless you use the /F option, NMAKE always searches for a file named MAKEFILE in the current directory.

NMAKE Options

NMAKE accepts a number of command-line options, which are listed in the following table.You may specify options in uppercase or lowercase letters and use either a slash or dash. For example, –B, /B, –b, and /b all represent the same option.

Option	Action
/A	Executes commands to build all the targets specified even if they are not out of date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from being displayed.
/D	Displays the modification date of each file when the dates of the target and dependents are checked.

/E	Causes environment macros to override macro definitions within description files. Normally, those macros defined in the description file that have the same names as environment macros cause a change in their value. The /E option tells NMAKE to ignore changes specified for these.
/F *filename*	Specifies *filename* as the name of the description file to use. If a dash (–) is entered instead of a filename, NMAKE accepts input from the standard input device instead of using a description file. If /F is not specified, NMAKE uses the file named MAKEFILE as the description file. If MAKEFILE does not exist, NMAKE uses the first string on the command line that is not an option or macro definition as the name of the file, provided the extension is not listed in the .SUFFIXES list (see the section "Pseudotargets" for more information).
/HELP	Displays help information about NMAKE syntax and options.
/I	Ignores exit codes (also called return or "errorlevel" codes) returned by programs called from the NMAKE description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays the commands from the description file that NMAKE would execute but does not execute these commands. This option is useful for checking which targets are out of date and for debugging description files.
/NOLOGO	Suppresses the NMAKE copyright message. Also see /C.
/P	Prints all macro definitions and target descriptions on the standard output device. This output can help you to debug your MAKEFILE.
/Q	Returns a zero status code if the target is up to date and a nonzero status code if it is not. This option is useful when invoking NMAKE from within a batch file.
/R	Ignores inference rules and macros contained in the TOOLS.INI file and also the predefined inference rules and macros.
/S	Does not display commands as they are executed.
/T	Changes the modification dates for out-of-date target files to the current date. The file contents are *not* modified.
/X *filename*	Sends all error output to *filename*, which can be either a file or a device. If a dash (–) is entered instead of a filename, the error output is sent to the standard output device.

Examples The following command line causes NMAKE to execute the commands in the description file QUICK to update the targets F1 and F2. The /C option prevents NMAKE from displaying nonfatal error messages and warnings.

```
NMAKE /F QUICK /C F1 F2
```

In the next example, NMAKE updates the target F1. If the current directory does not contain a file named MAKEFILE, NMAKE reads the file F1.MAK as the description file. The /D option displays the modification date of each file and the /N option displays the commands without executing them.

```
NMAKE  /D /N F1 F1.MAK
```

Invoking NMAKE Using a File

To invoke NMAKE with a file containing NMAKE command-line arguments, first create the file, then issue a command using the following syntax:

NMAKE @*commandfile*

Here *commandfile* is the name of a file containing the same information that would be specified on the command line: options, macro definitions, and targets. The command file is *not* the same as the description file.

Using a file to invoke NMAKE is useful when you have a long string of command-line arguments, such as macro definitions, that might exceed the DOS limit of 128 characters (256 for OS/2). Macro definitions can span multiple lines by ending each line except the last with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

For example, say that you have a file named UPDATE that contains the following lines:

```
/S "PROGRAM = FLASH" SORT.EXE SEARCH.EXE
```

To invoke NMAKE with the description file MAKEFILE, the /S option, the macro definition PROGRAM = FLASH, and the targets SORT.EXE and SEARCH.EXE, type the following:

```
NMAKE @UPDATE
```

Within the file, line breaks between arguments are treated as spaces. Macro definitions that contain spaces must be enclosed by quotation marks, just as if you typed them on the command line. You can continue a macro definition across multiple lines by ending each line except the last with a backslash (\) as in the following:

```
/S "PROGRAM \
= FLASH" SORT.EXE SEARCH.EXE
```

This file is equivalent to the first example. The backslash in the example allows the macro definition (`"PROGRAM = SAMPLE"`) to span two lines.

Description Files

NMAKE reads a description file to determine what to do. A description file is composed of one or more "description blocks," along with macros, inference rules, and directives. These can be in any order.

Each block indicates what files depend upon others and what commands need to be carried out to bring everything up to date.

When NMAKE runs, it builds the first target in the description file by default. You can override this default by specifying on the command line the names of the targets to build. The sections that follow describe the elements of a description file.

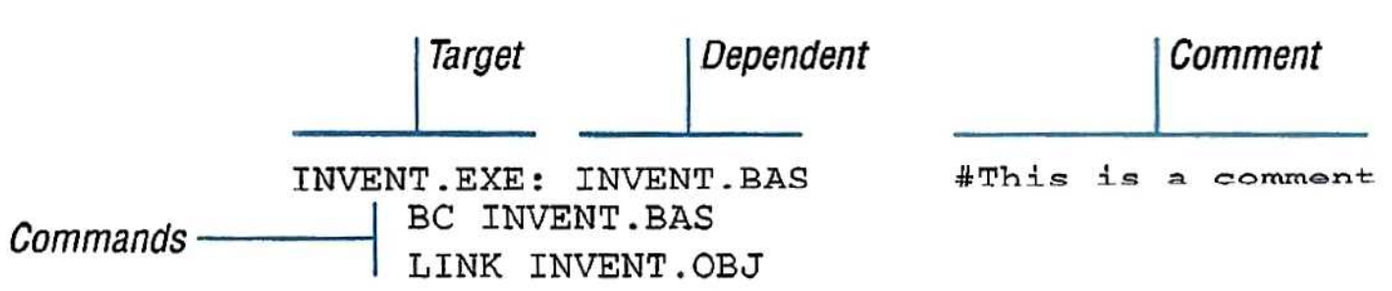
Description Blocks

An NMAKE description block has the following form:

target... : [[*dependent...*]] [[; *command*]] [[#*comment*]]
    [[*command*]]
[[#*comment*]]
[[{#*comment* | *command* }]]
    .
    .
    .

The file to be updated is *target; dependent* is a file upon which *target* depends; *command* is a command used to update *target*; and *comment* documents what is happening. The line containing *target* and *dependent* is called the dependency line.

For example, a simple description block is shown in the following figure:



Each component of a description block is discussed in the following table.

Component	Description
target	Specifies the name of one or more files to update. If you specify more than one file, separate the filenames by a space. The first target name must start in the first column of the line; it may not be preceded by any tabs or spaces. If the target name is one character long, then you need one space before the colon. For example, if your target is A, then `A :` is correct; `A:` will generate an error message. Note that the target need not be a file; it may be a pseudotarget, as described in "Pseudotargets," later in this chapter.
dependent...	The *dependent* field lists one or more files on which the target depends. If you specify more than one file, separate the filenames by a space. You can specify directories for NMAKE to search for the dependent files by using the following form: *target*: {*directory1*;*directory2*...}*dependent* NMAKE searches the current directory first, then *directory1*, *directory2*, and so on. If *dependent* cannot be found in any of these directories, NMAKE looks for an inference rule to create the dependent in the current directory. Inference rules are discussed later in this chapter. In the following example, NMAKE first searches the current directory for PASS.OBJ, then the \SRC\ALPHA directory, and finally the D:\PROJ directory: `FORWARD.EXE: {\SRC\ALPHA;D:\PROJ}PASS.OBJ`
command	The *command* is used to update the target. This can be any command that can be issued on the DOS or OS/2 command line. A semicolon must precede the command if it is given on the same line as the target and dependent files. Commands may be placed on separate lines following the dependency line, but each line must start with at least one space or tab character. Blank lines may be intermixed with commands. A long command may span several lines if each line ends with a backslash (\). If no commands are specified, NMAKE looks for an inference rule to build the target.
#comment	NMAKE considers any text between a number sign (#) and a new-line character to be a comment and ignores it. You may place a comment on a line by itself or at the end of any line except a command line. In the section of the description file that contains commands, comments must start in the first column.

Note If you are running OS/2 version 1.2, NMAKE will not accept target or dependent names beginning with a period (.).

Wildcard Characters

Certain utilities and programs accept DOS wildcard characters (* and ?) for specifying target and dependent filenames. NMAKE expands wildcards in target names when it reads the description file. It expands wildcards in the dependent names when it builds the target.

For example, suppose that you had three files, LIB1.BAS, LIB2.BAS, and LIB10.BAS. You could use a wildcard character to specify a group of files to be printed as follows:

```
PRINTLIB:
   !PRINT LIB*.BAS
```

PRINTLIB is a pseudotarget (described in the section "Pseudotargets" later in the chapter) that contains a command to print certain files. Since the DOS PRINT command accepts the * wildcard character, LIB1.BAS, LIB2.BAS, and LIB10.BAS would all be printed.

Note The BASIC Compiler (BC) does not accept wildcard characters to specify multiple filenames.

Escape Character

You can use a caret (^) to escape any DOS or OS/2 filename character in a description file. This can be useful if a character normally has a special meaning in a description file. The caret tells NMAKE to ignore the special meaning and have the character take on its literal value.

The following characters must be preceded by an escape character for NMAKE to interpret them literally:

( ) $ ^ \ ! @ -

For example, NMAKE interprets the specification "BIG^#.BAS" as the filename BIG#.BAS.

NMAKE ignores a caret that is not followed by any of the characters mentioned previously, as in the following:

```
MNO ^: DEF
```

In this case, NMAKE ignores the caret and treats the line as the following:

```
MNO : DEF
```

Carets that appear within quotation marks are not treated as escape characters.

Modifying Commands

Three different characters may be placed in front of a command to modify the command's effect. The character must be preceded by at least one space, and spaces may separate the character from the command. You may use more than one character to modify a single command. The characters are listed in the following table:

Character	Action

Dash (-)	Turns off error checking for the command. If the dash is followed by a number, NMAKE halts only if the error level returned by the command is greater than the number. In the following example, if the program FLASH returned an error code, NMAKE would not halt, but would continue to execute commands in the file:
	`LIGHT.LST:LIGHT.TXT`
	`  -FLASH LIGHT.TXT`
	Normally NMAKE would stop after the first error is encountered.

At sign(@)	Prevents NMAKE from displaying the command as it executes. In the following example, NMAKE does not display the ECHO command as it executes:
	`SORT.EXE:SORT.OBJ`
	`  @ECHO SORTING`
	The output of the ECHO command (that is, the message `SORTING`) would appear as usual.

Exclamation point (!)	Causes the command to be executed for each dependent file if the command uses one of the predefined macros $? or $**. The $? macro refers to all dependent files that are out of date with respect to the target, while $** refers to all dependent files in the description block. (See the section "Macros" later in this chapter for more information on predefined macros). For example:

```
PRINT: HOP.ASM SKIP.BAS JUMP.C
   !PRINT $** LPT1:
```

This example code causes the following three commands to be executed, regardless of the dependent file modification dates:

```
PRINT HOP.ASM LPT1:
PRINT SKIP.BAS LPT1:
PRINT JUMP.C LPT1:
```

Note that PRINT: is a pseudotarget (see "Pseudotargets" later in the chapter for rules governing these types of targets).

Specifying a Target in Multiple Description Blocks

You can specify more than one description block for the same target by using two colons (::) as the separator instead of one. For example:

```
TARGET.LIB :: A.ASM B.ASM C.ASM
   MASM A.ASM B.ASM C.ASM;
   LIB TARGET -+A.OBJ -+B.OBJ -+C.OBJ;
TARGET.LIB :: D.BAS E.BAS
   BC D.BAS;
   BC E.BAS;
   LINK D.OBJ E.OBJ;
   LIB TARGET -+D.OBJ -+E.OBJ;
```

These two description blocks update the library named TARGET.LIB. If any of the assembly language files have changed more recently than the library file, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the BASIC language files have changed, NMAKE executes the second group of commands, which compiles the BASIC files and then updates the library.

If you use a single colon in the preceding example, NMAKE issues an error message. You may, however, use single colons if commands are listed in only one block. In this case, dependency lines are cumulative. For example, consider these following commands:

```
TARGET: JUMP.BAS
   BC$?;
TARGET: UP.BAS
```

The preceding commands are equivalent to the following commands:

```
TARGET: JUMP.BAS UP.BAS
   BC $?;
```

How Description Blocks Work

A target is said to be out of date with respect to its dependents if any of the dependents have been modified since the last time the target was modified. NMAKE determines when a file has been modified by checking the last modification time stored by the operating system. If a target doesn't exist, then it is automatically out of date, regardless of the modification times of its dependents.

If NMAKE notes that a target is out of date, then the commands in the description block are executed. After these commands are executed, the modification date of the target is brought up to the current time and the target is no longer out of date. At this point NMAKE has "built" the target.

By default, NMAKE always builds the targets in the first description block of the description file. These targets are called the "primary targets." (You can override this default by specifying the names of other targets on the command line.)

If the dependent files of the primary targets are used as targets in other description blocks, then NMAKE ensures that these files are built before it builds the primary targets. In essence, NMAKE evaluates a "dependency tree." It builds files in the order necessary to bring the primary target up to date, never building a target until all files that depend on the target are up to date.

Macros

Macros provide a convenient way to replace a string in the description file with another string. The text is automatically replaced each time NMAKE is invoked. This feature makes it easy to change text used throughout the description file without having to edit every line that uses the text.

Macros can be used in a variety of situations, including the following:

- To create a standard description file for several projects. The macro represents the filenames used in commands. These filenames are then defined when you run NMAKE. When you switch to a different project, changing the macro changes the filenames NMAKE uses throughout the description file.
- To control the options that NMAKE passes to the compiler, assembler, or LINK. When you use a macro to specify the options, you can quickly change the options used throughout the description file in one easy step.

Macro Definitions

A macro definition uses the following form:

macroname = *string*

The *macroname* may be any combination of alphanumeric characters and the underscore (_) character; it is case sensitive. The macro name must start at the beginning of the line. Spaces before and after the equal sign are ignored. If a macro is defined more than once, then precedence rules govern which value is used.

You can define macros on the NMAKE command line or in the description file. Because of the way DOS parses command lines, the rules for the two methods are slightly different.

Defining Macros on the NMAKE Command Line

On the command line, no spaces may surround the equal sign. Spaces cause DOS to treat *macroname* and *string* as separate items. Strings that contain embedded spaces must be enclosed in double quotation marks. Alternatively, you can enclose the entire macro definition—*macroname* and *string*—in quotation marks. The *string* may be a null string.

Defining Macros in Description Files

In NMAKE description files, define each macro on a separate line. The first character of the macro name must be the first character on the line. NMAKE ignores spaces following *macroname* or preceding *string*. The *string* may be a null string or it may contain embedded spaces. Do not enclose *string* in quotation marks; NMAKE will consider them part of the string.

Using Macros

After you have defined a macro, use the following to include it in a dependency line or command:

$(*macroname*)

The parentheses are not required if *macroname* is only one character long. The *macroname* is converted to uppercase letters. If you want to use a dollar sign ($) in the file but do not want to invoke a macro, enter two dollar signs ($$), or use the caret (^) as an escape character preceding the dollar sign.

When NMAKE runs, it replaces all occurrences of $(*macroname*) with *string*. If the macro is undefined—that is, if its name does not appear to the left of an equal sign in the file or on the NMAKE command line—NMAKE treats it as a null string. Once a macro is defined, the only way to cancel its definition is to use the !UNDEF directive (see "Directives" later in this chapter for more about !UNDEF).

Example Assume the following text is in a file named MAKEFILE:

```
program = FLASH
bcopt = /ZI
linker = LINK
options = /CO /NOL /M
$(program).EXE : $(program).BAS
BC $(bcopt) $(program).BAS;
$(linker) $(options) $(program).OBJ;
```

When you invoke NMAKE, it interprets the description block as the following:

```
FLASH.EXE: FLASH.OBJ
   BC /ZI FLASH.BAS;
   LINK /CO /NOL /M FLASH.OBJ;
```

Note that NMAKE recognizes and processes the macro string exactly as you type it after the equal sign. Thus, "flash" and "FLASH" are not equivalent.

Macro Substitutions

Just as macros allow you to substitute text in a description file, you can also substitute text within a macro itself. Use the following syntax to do this:

$(*macroname*:*string1* = *string2*)

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. The colon must immediately follow *macroname*; however, spaces between the colon and *string1* are considered part of *string1*. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Example In the following example, the .BAS extension is substituted for the .OBJ extension for the macro SRCS:

```
SRCS = PROG.BAS SUB1.BAS SUR2.OBJ
PROG.EXE : $(SRCS:.BAS=.OBJ)
   LINK  $**;
DUP : $(SRCS)
   !COPY $** C:\BACKUP
```

Note that the special macro $** stands for the names of all the dependent files (see Table 20.1). The exclamation point (described in the section "Modifying Commands" later in this chapter) preceding the COPY command causes all dependent files in the description block (in this case PROG.BAS, SUB1.BAS, and SUB2.BAS) to be copied one at a time to the BACKUP directory.

If the preceding description file is invoked with a command line that specifies both targets, NMAKE will execute the following commands:

```
LINK PROJ.OBJ SUB1.OBJ SUB2.OBJ;
COPY PROJ.BAS C:\BACKUP
COPY SUB1.BAS C:\BACKUP
COPY SUB2.OBJ C:\BACKUP
```

The macro substitution does not alter the definition of the macro SRCS, but simply substitutes the extension .BAS with .OBJ. When NMAKE builds the target PROJ.EXE, the special macro $** (that is, the complete list of dependent files) is used, which specifies the macro substitution in SRCS. The same is true for the second target, DUP, except that no macro substitution is requested, so SRCS retains its original value and $** represents the names of the BASIC source files.

Predefined Macros

The following macro names are predefined:

Table 20.1 Predefined NMAKE Macros

Macro	Value
$*	The target name with the extension deleted.
$@	The full name of the current target.
$**	The complete list of dependent files that correspond to a target.
$<	The dependent file that is out of date with respect to the target (evaluated only for inference rules).
$?	The list of dependents that are out of date with respect to the target.

Table 20.1 *Continued*

Macro	Value
$$@	The target NMAKE is currently evaluating. This macro can only be used to specify a dependent. This macro is dynamic and has a different value for each target and so is called a "dynamic dependency parameter" for a target. See "Examples" in the next section for a typical use of this macro.
$(CC)	The command to invoke the C compiler. By default, NMAKE predefines this macro as CC = cl, which invokes the Microsoft C Optimizing Compiler. You might want to place the preceding definition in your TOOLS.INI file to avoid having to redefine it for each description file.
$(BC)	The command to invoke the Microsoft BASIC compiler, BC. NMAKE predefines this macro as BC = bc.
$(AS)	The command to invoke the Microsoft Macro Assembler. NMAKE predefines this macro as AS = masm.
$(MAKE)	The name with which the NMAKE utility was invoked. This macro is used to invoke NMAKE recursively. It causes the line on which it appears to be executed even if the /N option is on. You may redefine this macro if you want to execute another program; however, NMAKE returns a warning message.
$(MAKEFLAGS)	The NMAKE options currently in effect. If you invoke NMAKE recursively, you should use the command: $(MAKE) $(MAKEFLAGS). You cannot redefine this macro.

The macros $** and $? return one or more file specifications for the files in the target/dependent line of a description block. Except where noted, the file specification includes the path of the file, the base filename, and the filename extension.

Characters that Modify Predefined Macros

You can append characters to any of the first six macros in Table 20.1 to modify its meaning. Appending a D specifies the directory part of the filename only, an F specifies the filename, a B specifies just the base name, and an R specifies the complete filename without the extension. If you add one of these characters, you must enclose the macro name in parentheses. (The predefined macros $$@ and $** are the only exceptions to the rule that macro names longer than one character long must be enclosed in parentheses.)

For example, assume that $@ has the value C:\SOURCE\PROG\SORT.OBJ. The following table shows the effect the special characters have when combined with $@:

Macro	Value
$(@D)	C:\SOURCE\PROG
$(@F)	SORT.OBJ
$(@B)	SORT
$(@R)	C:\SOURCE\PROG\SORT

Examples

In following example, the macro $? represents the names of all dependents that are more recent than the target. The exclamation point causes NMAKE to execute the LIB command once for each dependent in the list. As a result of this description, the LIB command is executed up to three times, each time replacing a module with a newer version.

```
TRIG.LIB : SIN.OBJ COS.OBJ ARCTAN.OBJ
   !LIB TRIG.LIB -+$?;
```

The next example shows the use of NMAKE to update a group of include files. In the following description file, each of the files GLOBALS.H, TYPES.H, and MACROS.H in the directory C:\INCLUDE depends on its counterpart in the current directory. If one of the include files is out of date, NMAKE replaces it with the file of the same name from the current directory.

```
# Include files depend on versions in current directory
DIR=C:\INCLUDE
$(DIR)\GLOBALS.H : GLOBALS.H
   COPY GLOBALS.H $@
$(DIR)\TYPES.H : TYPES.H
   COPY TYPES.H $@
$(DIR)\MACROS.H : MACROS.H
   COPY MACROS.H $@
```

The following description file, which uses the special macro $$@, is equivalent:

```
# Include files depend on versions in current directory
DIR=C:\INCLUDE
$(DIR)\GLOBALS.H $(DIR)\TYPES.H $(DIR)\MACROS.H: $$(@F)
   !COPY $? $@
```

In this example, the special macro $$(@F) signifies the filename (without the directory) of the current target.

When NMAKE executes the description, it evaluates the three targets, one at a time, with respect to their dependents. Thus, NMAKE first checks whether C:\INCLUDE\GLOBALS.H is out of date compared with GLOBALS.H in the current directory. If so, it executes the command to copy the dependent file GLOBALS.H to the target. NMAKE repeats the procedure for the other two targets. Note that in the command line, the macro $? refers to the dependent for this target. The macro $@ means the full name of the target.

Macro Substitutions on Predefined Macros

You can perform macro substitutions on the following predefined macros:

$@

$*

$**

$?

$<

For example, suppose that you have this make file:

```
TARGET.ABC : DEPEND.XYZ
   ECHO $(@:TARG=BLANK)
```

If DEPEND.XYZ is out of date with respect to TARGET.ABC, then NMAKE executes the following command:

```
ECHO BLANKET.ABC
```

The example uses the predefined macro $@, which equals the full name of the current target. BLANK is substituted for TARG in the name BLANKET.ABC, resulting in BLANKET.ABC.

If you don't put a dollar sign in front of the predefined macro, the following is used:

```
$(@:TARG=BLANK)
```

This causes the line "ECHO (@:TARG=BLANK)" to be echoed to the screen. If you do use a dollar sign, the following is used:

```
$($@:TARG=BLANK)
```

This causes the full name of the current target to be echoed to the screen.

Precedence of Macro Definitions

If the same macro is defined in more than one place, the rule with the highest priority is used. The priority from highest to lowest is as follows:

1. Definitions on the command line
2. Definitions in the description file or in an include file
3. Definitions by an environment variable
4. Definitions in the TOOLS.INI file
5. Predefined macros such as CC and AS

If NMAKE is invoked with the /E option, macros defined by environment variables take precedence over those defined in a description file.

Macro Inheritance

When NMAKE starts, it creates macros equivalent to every current environment variable. These macros are called "inherited macros" because they have the same names and values as the corresponding environment variables, except that inherited macros are always in uppercase letters.

You can also reassign these inherited macros. For example, the following example modifies the PATH environment variable for the LINK command:

```
PATH=C:\TOOLS\BIN
TEST.EXE: TEST.OBJ
   LINK /CO TEST.OBJ
```

Regardless of what the PATH environment variable was set to previously, the macro variable PATH is set to the C:\TOOLS\BIN directory while NMAKE executes the commands in the make file. When NMAKE terminates, PATH is reset to its previous value.

Inference Rules

Inference rules are templates that NMAKE uses to generate files with a given extension. When NMAKE encounters a description block with no commands, it looks for an inference rule that specifies how to create the target from the dependent files, given the two file extensions. Similarly, if a dependent file does not exist, NMAKE looks for an inference rule that specifies how to create the dependent from another file with the same base name.

The use of inference rules eliminates the need to put the same commands in several description blocks. Inference rules have the following form:

.fromext.toext:
    command
    [[*command*]]
    .
    .
    .

The components of an inference rule are as follows:

Component	Description
fromext	Filename extension of the dependent required for the rule to be applied.
toext	Filename extension of the target that gets built when the rule is applied.
command	Set of commands used to build the *toext* target from the *fromext* target.

In the following example where BASIC source files are converted to object files, the inference rule looks like this:

```
.BAS.OBJ:
   BC /O $<;
```

The predefined macro $< represents the name of a dependent that is out of date relative to the target.

If NMAKE finds a description block without commands, it looks for an inference rule that matches both extensions. NMAKE searches for inference rules in the following order:

1. In the current description file.
2. In the tools-initialization file, TOOLS.INI. NMAKE first looks for the TOOLS.INI file in the current working directory and then in the directory indicated by the INIT environment variable. If it finds the file, NMAKE looks for the inference rules following the line that begins with the tag [NMAKE].

Only those extensions specified in the .SUFFIXES pseudotarget can have inference rules (see "Pseudotargets" later in the chapter for more information).

Note NMAKE applies an inference rule only if the base name of the file it is trying to create matches the base name of a file that already exists. In effect, this means that inference rules are useful only when there is a one-to-one correspondence between the files with the "from" extension and the files with the "to" extension. You cannot, for example, define an inference rule that inserts a number of modules into a library.

Specifying Paths

You can specify a single path for each extension, using the following form:

[[*frompath*]] .*fromext* [[*topath*]] . *toext* :
    commands

NMAKE takes the files with the *fromext* extension it finds in the directory specified by *frompath* and uses *commands* to create files with the *toext* extension in the directory specified by *topath*.

Predefined Inference Rules

NMAKE uses four predefined inference rules, which are summarized in Table 20.2.

Table 20.2 Predefined Inference Rules

Inference rule	Command	Default action
.bas.obj	$(BC) $(BFLAGS) $*.bas	BC $*.BAS
.c.obj	$(CC) $(CFLAGS) /c $*.c	CL /c $*.C
.c.exe	$(CC) $(CFLAGS) $*.c	CL $*.C
.asm.obj	$(AS) $(AFLAGS) $*;	masm $*;

Initially, BFLAGS, CFLAGS, and AFLAGS are undefined.

Example The following code defines an inference rule that executes the LINK command to create an executable file whenever a change is made in the corresponding object file:

```
.OBJ.EXE:
   LINK $<;

EXAMPLE1.EXE: EXAMPLE1.OBJ

EXAMPLE2.EXE: EXAMPLE2.OBJ
   LINK /CO EXAMPLE2,,,LIBV3.LIB
```

The filename in the inference rule is specified with the special macro $< so that the rule applies to any object file that has an out-of-date executable file.

When NMAKE does not find any commands in a description block, it checks for a rule that may apply and finds the rule defined on the first two lines of the description file. NMAKE applies the rule, replacing the $< macro with EXAMPLE1.OBJ when it executes the command, so that the LINK command becomes the following:

```
LINK EXAMPLE1.OBJ;
```

NMAKE does not search for an inference rule when examining the second description block because a command is explicitly given.

Directives

Using directives, you can construct description files that are similar to batch files. NMAKE provides directives that specify conditional execution of commands, display error messages, include the contents of other files, and turn on or off some of NMAKE's options.

Each directive begins with an exclamation point (!) in the first column of the description file. Spaces can be placed between the exclamation point and the directive keyword. The following table describes the directives:

Directive	Description
!IF *constantexpression*	Executes the statements between the !IF keyword and the next !ELSE or !ENDIF directive if *constantexpression* evaluates to a nonzero value.
!ELSE	Executes the statements between the !ELSE and !ENDIF directives if the statements preceding the !ELSE directive were not executed.
!ENDIF	Marks the end of the !IF, !IFDEF, or !IFNDEF block of statements.
!IFDEF *macroname*	Executes the statements between the !IFDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is defined in the description file. NMAKE considers a macro with a null value to be defined.
!IFNDEF *macroname*	Executes the statements between the !IFNDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is not defined in the description file.
!UNDEF *macroname*	Marks *macroname* as being undefined in NMAKE's symbol table.
!ERROR *text*	Causes *text* to be printed and then stops execution.

!INCLUDE *filename*	Reads and evaluates the file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise it looks in the current directory only. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.
!CMDSWITCHES {+|–}*opt*...	Turns on or off one of four NMAKE options: /D, /I, /N, or /S. If no options are specified, the options are reset to the way they were when NMAKE was started. Turn an option on by preceding it with a plus sign (+) or turn it off by preceding it with a minus sign (–). Using this directive updates the MAKEFLAGS macro.

The constant expression used with the !IF directive may consist of integer constants, string constants, or program invocations. Integer constants can use the C unary operators for numerical negation (–), one's complement (~), and logical negation (!). They may also use any of the C binary operators listed in the following table:

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^ ^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

You can use parentheses to group expressions. Values are assumed to be decimal values unless specified with a leading 0 (octal) or leading 0x (hexadecimal). Use the equality (==) operator to compare two strings for equality or the inequality (!=) operator to compare for inequality. Strings are enclosed by quotes. Program invocations must be in square brackets [ ].

Example The following example illustrates the use of the !INCLUDE, !CMDSWITCHES, !ERROR and !IFDEF/!ELSE/!ENDIF directives:

```
!INCLUDE <INFRULES.TXT>
!CMDSWITCHES +D
WINNER.EXE:WINNER.OBJ
!IFDEF debug
!   IF "$(debug)"=="y"
      LINK /CO WINNER.OBJ;
!   ELSE
      LINK WINNER.OBJ;
!   ENDIF
!ELSE
!   ERROR Macro named debug is not defined.
!ENDIF
```

The !INCLUDE directive causes the file INFRULES.TXT to be read and evaluated as if it were a part of the description file. The !CMDSWITCHES directive turns on the /D option, which displays the dates of the files as they are checked. If WINNER.EXE is out of date with respect to WINNER.OBJ, the !IFDEF directive checks to see if the macro debug is defined. If it is defined, the !IF directive checks to see if it is set to `y`. If it is, then LINK is invoked with the /CO option; otherwise it is invoked without. If the debug macro is not defined, the !ERROR directive prints the message and NMAKE stops executing.

Pseudotargets

A "pseudotarget" is a target that is not a file but instead is a name that serves as a "handle" for building a group of files or executing a group of commands. In the following example, `UPDATE` is a pseudotarget:

```
UPDATE: *.*
   !COPY $** A:\PRODUCT
```

When NMAKE evaluates a pseudotarget, it always considers the dependents to be out of date. In the preceding description, NMAKE copies each of the dependent files to the specified drive and directory.

The NMAKE utility includes four predefined pseudotargets that provide special rules within a description file. The following table describes these pseudotargets:

Pseudotarget	Action
.SILENT:	Does not display lines as they are executed. Same effect as invoking NMAKE with the /S option.
.IGNORE:	Ignores exit codes returned by programs called from the description file. Same effect as invoking NMAKE with the /I option.

.SUFFIXES:*list*	Lists file extensions for NMAKE to try if it needs to build a target file for which no dependents are specified. NMAKE searches the current directory for a file with the same name as the target file and a suffix from the list. If NMAKE finds such a file, and if an inference rule applies to the file, then NMAKE treats the file as a dependent of the target. The order of the suffixes in the list defines the order in which NMAKE searches for the files. The list is predefined as follows:
	`.SUFFIXES: .OBJ .EXE .C .ASM .BAS`
	To add extensions to the list, specify .SUFFIXES: followed by the new suffixes. For example, the following appends the .XYZ extension to the previous value of .SUFFIXES:
	`.SUFFIXES: .XYZ`
	To clear the list, specify the following:
	`.SUFFIXES:`
	Only those extensions specified in .SUFFIXES can have inference rules.

.PRECIOUS: *target...*	Tells NMAKE not to delete *target* if the commands that build it are quit or interrupted. Using this pseudotarget overrides the NMAKE default. By default, NMAKE deletes the target if it cannot be sure the target was built successfully.

For example:

```
PRECIOUS: TOOLS.LIB  TOOLS.LIB : A2Z.OBJ Z2A.OBJ
          .
          .
          .
```

If the commands (not shown here) to build TOOLS.LIB are interrupted, leaving an incomplete file, NMAKE does not delete the partially built TOOLS.LIB because it is listed with .PRECIOUS.

Note, however, that .PRECIOUS is useful only in limited circumstances. Most professional development tools, including those provided by Microsoft, have their own interrupt handlers and "clean up" when errors occur.

The TOOLS.INI File

You can customize NMAKE by placing commonly used macros and inference rules in the TOOLS.INI initialization file. Settings for NMAKE must follow a line that begins with [NMAKE]. This part of the initialization file can contain macro definitions, .SUFFIXES lists, and inference rules.

If TOOLS.INI contains the following section, for example, NMAKE reads and applies the lines following the tag [NMAKE]. The example defines the macros PROGRAM, BCOPT, and OPTIONS, and adds the extension .BAS to the .SUFFIXES list.

```
[NMAKE]
PROGRAM = FLASH
BCOPT = /ZI
OPTIONS = /CO /M /G2
.SUFFIXES: .BAS
```

NMAKE looks for TOOLS.INI in the current directory. If it is not found there, NMAKE searches the directory specified by the INIT environment variable.

In-Line Files

NMAKE can generate "in-line files," which can contain any text you specify. A common use of in-line files is to generate a response file for another utility such as LIB.

Use this syntax to tell NMAKE to create an in-line file:

target : *dependents*
  command @<<[[*filename*]]
inlinetext
<<[[{KEEP | NOKEEP}]]

All of the text between the two sets of angle brackets (<<) is placed in a file. The second pair of angle brackets must be at the beginning of the line, with no preceding white-space characters. Note that the at sign (@) is not part of the NMAKE syntax but is the typical response-file character for utilities such as LINK and LIB.

To name the in-line file, specify a filename immediately after the first pair of angle brackets, with no intervening spaces. If you do not specify a filename, NMAKE gives the response file a unique name. NMAKE places the file in the directory specified by the TMP environment variable if the variable is defined; if the TMP variable is not defined, NMAKE creates the response file in the current directory.

For example, the following in-line file creates a LIB response file called LIB.LRF, then invokes LIB:

```
MATH.LIB : ADD.OBJ SUB.OBJ MUL.OBJ DIV.OBJ
   LIB @<<LIB.LRF
MATH.LIB
-+ADD.OBJ-+SUB.OBJ-+MUL.OBJ-+DIV.OBJ
LISTING
<<
```

The resulting response file, LIB.LRF, specifies which library to use, the commands to execute, and the listing file to produce:

```
MATH.LIB
-+ADD.OBJ-+SUB.OBJ-+MUL.OBJ-+DIV.OBJ
LISTING
```

Keeping In-Line Files

The KEEP and NOKEEP keywords tell NMAKE whether or not to delete an in-line file. Unless you specify KEEP, NOKEEP is used and the in-line file is deleted.

For example, the following prevents NMAKE from deleting the file BIG.LRF:

```
BIG.EXE: $(OBJS)
   link @<<BIG.LRF
$(OBJS);
<<KEEP
```

By default, NOKEEP is used.

Using Whitespace Characters

You may use literal whitespace characters in macro substitutions. For example, suppose that you want to create a LINK response file that specifies a long list of object files you want to link:

```
OBJS=AA.OBJ AB.OBJ ...\
    .
    .
    .
    ZZ.OBJ
```

Normally, the $(OBJS) macro would overflow the limit of LINK (127 bytes per line for DOS, 256 bytes per line for OS/2). To prevent this from happening, you can substitute the plus (+) and new-line characters, (the caret is used as the new-line character) for each whitespace in the file as follows:

```
BIG.EXE: $(OBJS)
   LINK @<<BIG.LRF
$(OBJS: = +^
)
BIG
/CO;
<<
```

This places each object module on a separate line, thus preventing an overflow condition. The next two fields represent the executable file (BIG) and LINK options (/CO) you want in the response file. To accept LINK's default values for the remaining fields, type a semicolon at the end of the last line you are specifying.

Creating More than One In-Line File

A description block can create more than one in-line file at once. For example, the following example creates two in-line files named FILE1 and FILE2:

```
TARGET.XXX : DEPEND.YYY
   CAT <<FILE1 <<FILE2
I am the contents of FILE1.
<<KEEP
I am the contents of FILE2.
<<KEEP
```

This causes the following command to be executed:

```
CAT FILE1 FILE2
```

The KEEP keywords tell NMAKE not to delete FILE1 and FILE2 when done.

Sequence of Operations

If you are writing a complex description file, you may need to know the exact order of steps that NMAKE follows. This section describes those steps in order.

When you run NMAKE from the command line, its first task is to find the description file, following these steps:

1. If NMAKE is invoked with the /F option, it uses the filename specified in the option.
2. If /F is not specified, NMAKE looks for a file named MAKEFILE in the current directory. If such a file exists, it is used as a description file.
3. If MAKEFILE is not in the current directory, NMAKE searches the command line for the first string that is not an option or a macro definition, and treats this string as a filename. If the filename extension does not appear in the .SUFFIXES list, NMAKE uses the file as the description file. If the extension appears in the .SUFFIXES list, NMAKE tries additional strings until it finds a suitable file. (See the section "Pseudotargets" earlier in this chapter for more about the .SUFFIXES list).
4. If NMAKE still has not found a description file, it returns an error.

NMAKE stops searching for a description file as soon as it finds one, even if other potential description files exist. If you specify /F, NMAKE uses the file specified by that option even if MAKEFILE exists in the current directory. Similarly, if NMAKE uses MAKEFILE, then any description file listed in the command line is treated as a target.

NMAKE then applies macro definitions and inference rules in the following order:

1. Predefined macros and inference rules
2. Macros and inference rules defined in TOOLS.INI
3. Macros defined by an environment variable
4. Macros and inference rules defined in the description file or in an included file
5. Macros defined on the command line

Definitions in later steps take precedence over definitions in earlier steps. The /E option, however, causes macros defined by environment variables to override macros defined on the command line. The /R option causes NMAKE to ignore the contents of TOOLS.INI and the predefined inference rules and macro definitions.

Now NMAKE updates each target in the order in which it appears in the description file. It compares the date and time of each dependent with those of the target, and executes the commands needed to update the target. If you specify the /A option, or if the target is a pseudotarget, NMAKE updates the target even if its dependents are not out of date.

If the target has no explicit dependents, NMAKE looks in the current directory for one or more files whose extensions are in the .SUFFIXES list. If it finds such files, NMAKE treats them as dependents and updates the target according to the commands.

If no commands are given to update the target, or if the dependents cannot be found, NMAKE applies inference rules to build the target. By default, it tries to build executable files from object files; and it tries to build object files from .C and .ASM sources.

NMAKE normally quits processing the description file when a command returns an error. In addition, if it cannot tell that the target was built successfully, NMAKE deletes the partially created target. If you use the /I command-line option, NMAKE ignores exit codes and attempts to continue processing. The .IGNORE pseudotarget has the same effect. To prevent NMAKE from deleting the partially created target, specify the target name in the .PRECIOUS pseudotarget.

Alternatively, you can use the dash command modifier (–) to ignore the error code for an individual command. An optional number after the dash tells NMAKE to continue if the command returns an error code that is less than or equal to the number and to stop if the error code is greater than the number.

You can document errors by using the !ERROR directive to print descriptive text. The directive causes NMAKE to print some text, then stop, even if you use /I, .IGNORE, or the dash modifier.

Differences Between NMAKE and MAKE

NMAKE differs from MAKE in the following ways:

- It accepts command-line arguments from a file.
- It provides more command-line options.
- It does not evaluate targets sequentially, as MAKE does. Instead, it updates the targets specified on the command line, regardless of where they appear in the description file. If no targets are specified, NMAKE updates the first target in the file.
- It provides more predefined macros.
- It permits substitutions within macros.
- It supports directives placed in the description file.
- It allows you to specify include files in the description file. MAKE assumed that all targets in the description file would be built. Because NMAKE builds the first target in the file unless you specify otherwise, you may need to change your old description files to work with the new utility. Description files written for use with MAKE typically list a series of subordinate targets followed by a higher-level target that depends on the subordinates. As MAKE executed, it would build the targets sequentially, creating the highest-level target at the end.

The easiest way to convert these description files is to create a new description block at the top of the file. Give this block a pseudotarget named ALL and set its dependents to all of the other targets in the file. When NMAKE executes the description, it will assume you want to build the target ALL and consequently will build all targets in the file. Alternatively, if your description file already contains a block that builds a single, top-level target, you can simply make that block the first in the file.

Example The following is an old MAKE description file named MAKEFILE. Note that it builds two top-level targets, PROG1.EXE and XYZ.EXE.

```
ONE.OBJ: ONE.BAS
TWO.OBJ: TWO.BAS
THREE.OBJ: THREE.BAS
PROG1.EXE: ONE.OBJ TWO.OBJ THREE.OBJ
   LINK ONE TWO THREE, PROG1;
X.OBJ: X.BAS
   BC /ZI X.BAS;
Y.OBJ: Y.BAS
   BC /ZI Y.BAS;
Z.OBJ: Z.BAS
   BC /ZI Z.BAS;
XYZ.EXE: X.OBJ Y.OBJ Z.OBJ
   LINK X Y Z, XYZ;
```

To use this file with the new NMAKE, insert the following as the first line in the file:

```
ALL : PROG1.EXE XYZ.EXE
```

With the addition of this line, `ALL` becomes the first target in the file. Since NMAKE, by default, builds the first target, you can build both PROG1.EXE and XYZ.EXE by typing the following:

```
NMAKE
```

Chapter 21
Building Custom Run-Time Modules

This chapter explains how to create and use a custom run-time module. Microsoft BASIC lets you embed your own routines into the BASIC run-time module itself, which lets you create a custom version of the default run-time module. This feature allows you to put often used routines in the BASIC run-time module where they can be used by any application that you create. Such modules streamline the development process and minimize the size of the executable file for each separate application. This feature is designed for programs that require the presence of the Microsoft BASIC run-time module when they are run (that is, programs compiled without the /O option).

For instance, say that you are a professional developer who markets several data-processing applications, and you have written a general-purpose sort routine that is called by all of your applications. If you embed your sort routine in a custom run-time module, it is instantly available to any of your applications at run time, exactly like the standard BASIC run-time routines. When providing an update of one of your products, you can ship only that application's executable file, which will be considerably smaller because it does not contain the code for your sort routine or the routines in the BASIC run-time module.

You can build custom run-time modules for either protected mode or real mode. Custom run-time modules are particularly useful in protected mode because the code they contain is dynamically linkable.

Creating a Custom Run-Time Module

Here are the basic development steps for creating an application that uses a custom run-time module:

1. Compile the source files containing the routines that you will add to the run-time module.
2. Create an export list file that lists the object files you will add to the run-time module, the names of all of the routines you wish to call from those objects, and libraries you want to withdraw code from.
3. Run the BUILDRTM utility to create the custom run-time module and its support files (the run-time module library and IMPORT.OBJ file).
4. Link the object files for your application with the support files for your custom run-time module.

The following sections explain these steps in detail.

Compiling Source Files

The first step in creating a custom run-time module is to compile the source files containing the code you wish to add to the run-time module. Note that the resulting object files must be compiled correctly for the target environment (real or protected mode), floating-point method, and memory model (near or far strings) that your final application will use. See Chapter 16, "Compiling With BC," for details about options to BASIC Compiler (BC).

Creating an Export List

After compiling your source file into an object file, you must create an "export list," which is a text file containing the following information:

- A list of the modules you will add to the run-time module.
- A list of the procedures you want to call from the preceding modules.
- A list of libraries that contain routines referred to by your objects (for example, the C run-time library).

You can create the export list with any editor that creates ASCII text files. The purpose of the list is to indicate which user-created modules you will add to the run-time module, and which routines in those modules you expect to call from a program. This information is later used by the BUILDRTM utility (described later in this chapter) to add your modules to the run-time module and make your routines accessible to an application.

You may put explanatory comments in the export list. A comment line must begin with a number sign (#); blank lines in the export list are ignored. Here is a simple export list:

```
#  Comment lines begin with the '#' character.
#  The following list shows which user-created
#  modules to add to the BASIC run-time module.
#  These names may include a path.

#OBJECTS
SAMPLE1
SAMPLE2
NOGRAPH.OBJ

#  The following list names all of the SUB and
#  FUNCTION procedures that you want to call from the
#  preceding modules. To be called from a program,
#  the procedure's name must appear on this list.

#EXPORTS
SubFirst
SubSecnd
FuncOne
FuncTwo
#LIBRARIES
SAMPLE.LIB
```

The export list uses three directives, OBJECTS, EXPORTS, and LIBRARIES. These directives must be preceded by a number sign (#) without any intervening spaces, as shown in the preceding example. You can list these directives in any order or combination in your export list file. The same directive may appear more than once in your file (this allows you to concatenate two files without modification).

OBJECTS Directive

The OBJECTS directive must be followed by a list of the modules that you wish to include in the custom run-time module. This list of filenames comes immediately after the OBJECTS directive. You need list only the filename of the module; it is assumed that the file has an .OBJ extension, (however, if you supply an extension, that extension overrides the default extension).

You can also place stub files, such as NOGRAPH.OBJ, in this list. These files will add or remove code, normally included during linking, that supports certain features. For example, if you include the NOGRAPH.OBJ file after the #OBJECTS directive, code that supports graphics features will be excluded. For more about stub files, see Chapters 18, "Using LINK and LIB," and 15, "Optimizing Program Size and Speed."

You may include a path in the specification for any file in this list. For example, both of these file specifications are valid within an export list:

```
C:\OS2DEV\SRC\MYOBJECT
..\..\SRC\MYOBJECT.OBJ
```

EXPORTS Directive

The EXPORTS directive must be followed by a list of the names of the **SUB** and **FUNCTION** procedures that you want to use from the modules listed in the OBJECTS section of the export list. These are the names of the procedures that you wish to invoke from your final application. (You need not list routines that are only used internally.) You may list up to 255 procedures.

LIBRARIES Directive

The LIBRARIES directive is followed by a list of library file names BUILDRTM will search when creating a custom run-time module. This allows you to add routines contained in an existing library to your custom run-time module. For example, if SAMPLE1.OBJ makes calls to the function MyFunc, which is found in the library SAMPLE.LIB, you must list SAMPLE.LIB under the LIBRARIES directive.

Invoking the BUILDRTM Utility

The next step in building a custom run-time module is to invoke the BUILDRTM utility. When you invoke BUILDRTM, it reads the export list, evaluates the environment options, if any, and builds the customized run-time module and support files.

BUILDRTM calls on several different disk files and utilities. The /HELP option causes BUILDRTM to display all of the options that it accepts and a list of the files that it may require. You must make sure that BUILDRTM can find all of the files that it needs, depending on your choice of target environment, floating-point method, and memory/string model. This can be done by setting the PATH environment variable to reference the executable files and the LIB environment variable to reference the libraries needed, by specifying explicit pathnames on the command line, or by moving files into the current directory.

The program BUILDRTM.EXE is a bound application, meaning that it can be run in both protected mode and real mode. The syntax for invoking BUILDRTM is as follows:

BUILDRTM [[*options*]] {/DEFAULT | *runtime exportlist*}

The *runtime* argument is a string containing the filename you wish the custom run-time module to have, and *exportlist* is a string containing the file name of your export list.

Possible values for *options* are as follows:

Option	Description
/H[[ELP]]	Displays help about BUILDRTM options and required files.
/FP*method*	Specifies the floating-point method used. Options are /FPa (alternate math), /FPi (in-line coprocessor support or emulation), and /FPi87(math-coprocessor required library). The default is /FPi. Before building a custom run-time module, make sure all modules contained in it have been compiled using the same /FP*method* option of BC. You should only use the /FPi87 option if the custom run-time module will always be used on a machine having a math coprocessor. See Chapter 16, "Compiling With BC," for more information about floating-point options.
/L*mode*	Specifies the target environment. Options are /LR (DOS or real mode) and /LP (protected mode). If you don't supply this option, BUILDRTM creates a custom run-time module suitable for the environment you are in at the time BUILDRTM is invoked (real or protected mode).
/Fs	Enables far string support (more than 64K). If /Fs is specified, far string support is enabled; otherwise, near string support is assumed.
/MAP	Generates a full map file for the custom run-time module.

Building Default Run-Time Modules and Libraries

BUILDRTM can also be used to create default run-time modules and libraries. When you run the Setup program to install Microsoft BASIC, Setup invokes BUILDRTM to create default run-time modules and corresponding run-time module combined libraries. After installation, you may choose to invoke BUILDRTM with the /DEFAULT option to build additional run-time modules and run-time-module libraries that you did not create during the setup process, or to recreate these files if they have been accidentally deleted.

To see which default run-time module and libraries can be created, type BUILDRTM /H, and view the section "Default output files" (see the section "Types of Libraries" in Chapter 17, "About Linking and Libraries," for naming conventions used with run-time modules and libraries).

BRT70 *float string mode*.EXE (Real mode)

BRT70 *float string mode*.DLL (Protected mode)

And run-time libraries use the following naming convention:

BRT70 *float string mode*.LIB

The possible values for each variable are shown in the following table:

Variable	Letter	Feature
float	E	Emulator (/FPi)
	A	Alternate math (/FPa)
string	N	Near strings
	F	Far strings (/Fs)
mode	R	Real mode(/LR)
	P	Protected mode (/LP)

For example, if you specified protected mode, in-line instructions, and far string support during Setup, the BRT70EFP.DLL run-time module and BRT70EFP.LIB library would be created.

Run-Time Messages from BUILDRTM

In the course of its work, BUILDRTM calls upon three other supplied utilities: LINK, LIB, and IMPLIB (OS/2 run-time modules only). To inform you of its progress, BUILDRTM displays informational messages as the build progresses.

If an error occurs while BUILDRTM has invoked one of these utilities, the invoked utility issues its own error message and BUILDRTM terminates. See Chapters 17, "About Linking and Libraries," and 18, "Using LINK and LIB," for more information about the LINK and LIB utilities.

Results from BUILDRTM

The BUILDRTM utility creates the following three files:

- An import object file named IMPORT.OBJ

  BUILDRTM always creates an "import object file" named IMPORT.OBJ. This file contains information that allows your application to access data. You must link this file whenever you create an application that uses a custom run-time module. Although it always has the same name upon creation, this file is specific to the run-time module for which it was made. Thus, if you create more than one custom run-time module, you may want to rename each IMPORT.OBJ file with a name that clearly associates it with its parent run-time module.

- A run-time library

  BUILDRTM creates the "run-time library" for your custom run-time module. This file makes it possible for LINK to resolve references to the routines and data objects in the modules that you add to the run-time module. The run-time library has the filename that you supplied when you invoked BUILDRTM; like other libraries, it has the extension .LIB.

- A custom run-time module

  BUILDRTM also creates the custom run-time module itself. The custom run-time module has the filename that you supplied when you invoked BUILDRTM (it shares the same filename as the run-time library). Its filename extension matches the target operating environment — either .EXE for real mode or .DLL for protected mode.

Note You may rename the IMPORT.OBJ file and the custom run-time module library. However, you must not rename the custom run-time module after it has been created by BUILDRTM. If you change the name of the custom run-time module, your application will not be able to find it at run time.

Creating Executable Files

Once you have successfully run BUILDRTM, your custom run-time module is complete. You can create many different applications that call the custom routines you have embedded in the BASIC run-time module. Here is the LINK command you can use to create such an application:

```
LINK IMPORT.OBJ objects..., executable, , runtime.LIB/NOD;
```

The *objects* field stands for the object file or files that you have created for this specific application; *executable* is the name you wish to give the final executable file; and *runtime*.LIB is the name of the run-time module library that BUILDRTM created for your custom run-time module.

For example, the following command line links your object module MYOBJ.OBJ into the run-time module SAMPLE.EXE:

```
LINK IMPORT.OBJ MYOBJ.OBJ, SAMPLE.EXE, , SAMPLE.LIB/NOD;
```

Important The IMPORT.OBJ file must be the first item on the command line, after LINK command.

You must also supply the /NOD option to prevent LINK from linking the default BASIC run-time module library.

Programming Considerations for Custom Run-Time Modules

Certain common-sense precautions are required to ensure that your custom run-time module works as intended. As mentioned earlier, you must take care that all of the modules that make up an application share the same target environment, floating-point method and string model. An inconsistency of this type inevitably causes LINK errors or a run-time failure.

Handling Data Objects Consistently

You must ensure that all of the modules in a given application define and handle data objects consistently. For instance, if a routine embedded in a custom run-time module defines data objects as **COMMON**, routines in your application must not attempt to define the same object as **COMMON** with a larger size (this rule applies to both named and unnamed **COMMON** definitions). This sort of inconsistency may corrupt an application's data.

Assembly Language Programs

If you are writing in assembly language, you must make sure that all calls to routines in a custom run-time module are true far calls. That is, you should not call any routine indirectly through a variable, or perform a pseudocall by pushing a return address onto the stack and executing an unconditional jump.

A second precaution for assembly language programmers has to do with near-data declarations. Any near data that you add to a custom run-time module must be defined to be of the classes CONST, DATA, or BSS rather than some unknown class. This should not be a problem if you use the simplified segment directives supported by the MASM versions 5.0 and higher, since those directives conform to this rule. The same is true of all high-level Microsoft languages. In other words, this precaution is important only if you use "old-style" MASM segment declarations rather than the simplified segment directives supported by MASM versions 5.0 and higher. The following old-style data declarations, for instance, can be used to define near data in a custom run-time module:

```
myseg1 SEGMENT BYTE PUBLIC 'CONST'
myseg1 ENDSmyseg2 SEGMENT WORD PUBLIC 'DATA'
myseg2 ENDSmyseg3 SEGMENT PARA COMMON 'BSS'
myseg3 ENDSDGROUP GROUP myseg1, myseg2, myseg3
```

The following declarations should not be used because they define near data in the default data segment as an unknown class (the class STRANGECLASS) and as no class:

```
myseg1 SEGMENT BYTE PUBLIC 'STRANGECLASS'
myseg ENDSmyseg2 SEGMENT
mseg2 ENDSDGROUP GROUP myseg1, myseg2
```

DGROUP References

For mixed-language programs that use the **CHAIN** statement, you should make sure that any code built into a custom run-time module does not contain references to DGROUP. **CHAIN** causes DGROUP to move, but does not update references to DGROUP. So the chained program that references DGROUP would reference the data in the current DGROUP segment. For example, the following instruction should not be used in an assembly language program that is to be included in a custom run-time module:

```
mov    ax,@data
```

This rule applies only to mixed-language programs. You can ignore this warning if your program contains only routines written in BASIC, because BASIC routines never refer to DGROUP.

Chapter 22
Customizing Online Help

The Microsoft Help File Creation Utility (HELPMAKE) allows you to create or modify Help files for use with Microsoft products. For example, you could write Help text describing a new function you have written for a Quick library. This would let you display Help information about your function anytime you're working in the QBX environment.

HELPMAKE translates Help text files into a Help database accessible from within the following:

- The QBX environment
- Microsoft QuickHelp utility
- Other Microsoft language products

To use HELPMAKE, you need one or more input files that contain information specially formatted for the Help system. Specify the name of a Help text file formatted in one of several simple styles and the amount by which to compress the file. HELPMAKE can also decompress a Help database to its original text format.

What's in a Help File?

This section defines some of the terms involved in formatting and outlines the types of files HELPMAKE can use as input. A Help database file can contain the following elements:

- Contexts and topic text
- Implicit cross-references
- Explicit cross-references
- Formatting flags

Each of these elements is discussed in the following sections.

Contexts and Topic Text

If you have used the online Help system in QBX you probably have a good idea of what a Help file looks like. As you might expect, each file starts with a subject and some information about the subject, then lists another subject and some information about it, then another, and so on. In HELPMAKE terminology, the subjects are called "contexts" and the information is called "topic text." The application simply hands the context to an internal Help engine, which searches the database for information.

Whenever someone asks for Help on the **OPEN** statement from within the QBX environment, for example, the Microsoft Advisor Online Help System looks for the context "open" and displays its topic text. (The name of every function in the BASIC statement and function library is a context throughout the Help database.)

Often the same information applies to more than one subject. For example, the BASIC functions **CVI**, **CVS**, **CVL**, and **CVD** are all contexts for one block of topic text. The converse, however, is not true. You cannot specify different blocks of topic text, in different places in the Help file, to describe a single subject.

Cross-References

To make it easier for users to navigate through a Help database, you can put cross-references in your Help text. Cross-references bring up information on related topics, including header files and code examples. The Help for the **PRINT** statement, for example, refers to the **PRINT USING** and **PRINT#** statements. Cross-references can point to other contexts in the same Help database, to contexts in other Help databases, or to ASCII files outside the database.

Help files can have two kinds of cross-references:

- Implicit cross-references
- Explicit cross-references, or hyperlinks

Implicit Cross-References

The word "open" is an implicit cross-reference throughout the QBX Help database because it is the name of a BASIC function. If a user selects the word "open" anywhere in QBX or Help, the Help system displays information on the **OPEN** function. Cross-references like this are called implicit cross-references because they are implicit in the Help file and require no special coding. Anywhere a context appears, the Help system makes an implicit cross-reference to its topic text.

Explicit Cross-References

Explicit cross-references, also called "hyperlinks," are tied to a word or phrase at a specific location in the Help file. You set up explicit cross-references when you write the Help text. For example, suppose that you want to display Help on the **PRINT** statement at a particular position in the QBX Help system if the user selects the word "formatting." You would create an explicit cross-reference from the word "formatting" to the context **PRINT**. Anywhere else in the file, "formatting" would have no special significance, but at that one position, it would display the topic text for **PRINT**.

Formatting Flags

Help text can also include formatting flags to control the appearance of the text on the screen. Using these flags, you can make certain words appear in boldface, others underlined, and so on, depending on the graphics capabilities of the user's computer.

In the Help database Microsoft supplies for QBX, the active cross-references are surrounded by green symbols on a color monitor. Other applications may represent cross-references differently; for example, in bold or in color.

Specific formatting flags for QuickHelp format and Rich Text Format are discussed in "Using-Help Database Formats" later in this chapter.

Help File Formats

You can create Help files using any of three formats:

- QuickHelp format
- Rich Text Format (RTF)
- Minimally formatted ASCII

In addition, you can reference unformatted ASCII files, such as include files, from within a Help database.

QuickHelp

QuickHelp format is the default and is the format in which HELPMAKE writes files it decodes from existing Help databases. Use any text editor to create a Help text file in the QuickHelp format. The QuickHelp format also lends itself to a relatively easy automated translation from other document formats.

QuickHelp files can contain all the various cross-references and formatting flags. For more information about QuickHelp formatting flags, see the section "QuickHelp Formatting Flags" later in this chapter. Typically, you would use QuickHelp format for any changes you want to make to the standard Help database. Most of the examples in this chapter are in QuickHelp format.

Note

The QuickHelp format supported by QBX is a subset of the format supported by the OS/2 QuickHelp utility. QBX does not support the complete set of OS/2 QuickHelp dot commands.

Rich Text Format

Rich Text Format (RTF) is a Microsoft word-processing format that many other word processors also support. You can create RTF Help text with any word processor capable of generating RTF output. You may also use any utility program that takes word-processor output and produces an RTF file.

Use RTF when you want to transfer Help files from one application to another while retaining formatting information. You can format RTF files directly with the word-processing program and need not edit them to insert any special commands or tags. Like QuickHelp files, RTF files can contain formatting codes and cross-references. For a list of the RTF formatting codes recognized by HELPMAKE, see the section "RTF" later in this chapter.

Minimally Formatted ASCII

Minimally formatted ASCII files define contexts and their topic text. They are intended for Help files less than 256K that do not require compression, and for access of README-type text directly by online Help. These files cannot contain explicit cross-references or formatting flags.

Unformatted ASCII

Unformatted ASCII files must be smaller than 64K and are exactly what their name implies: regular ASCII files with no special formatting commands or context definitions. An unformatted ASCII file does not become part of the Help database. Instead, only its name is used as the object of a cross-reference. For example, unformatted ASCII files could contain program examples.

Invoking HELPMAKE

HELPMAKE encodes or decodes Help files. Encoding is the process of converting a text file into a compressed Help database. Decoding reverses the process: it converts a Help database into a text file. HELPMAKE can decode any Microsoft Help database file (unless it was encoded with the /L option) to a QuickHelp formatted text file for editing. It can also encode an RTF, QuickHelp, or minimally-formatted ASCII text file into Help database format.

HELPMAKE is required to create and modify Microsoft-compatible Help databases. It is not required, however, merely to access databases supplied with Microsoft language products.

The HELPMAKE command-line syntax is as follows:

HELPMAKE [[*options*]] { /E [[*n*]] | /D [[*letter*]] } *sourcefiles*

The options are described in the next section. In addition, you can type HELPMAKE by itself or with the /H option to display a summary of valid HELPMAKE options.

Either the /E (encode) or the /D (decode) option must be supplied. When encoding (/E) to create a Help database, you must use the /O option to specify the filename of the database. The optional *n* parameter specifies the amount of compression to take place (described in "Options for Encoding" later in this chapter).

The *sourcefiles* field is required. It specifies the input files for HELPMAKE. If you use the /D (decode) option, *sourcefiles* may be one or more Help database files. HELPMAKE decodes the database files into a single text file. If you use the /E (encode) option, *sourcefiles* may be one or more Help text files. Separate filenames with a space. Standard wildcard characters may also be used.

Example The following HELPMAKE command line encodes a source file by invoking HELPMAKE with the /V, /E, and /O options:

```
HELPMAKE /V /E /OMY.HLP MY.TXT > MY.LOG
```

HELPMAKE reads input from the text file MY.TXT and writes the compressed Help database in the file MY.HLP. The /E option causes maximum compression. Note that the DOS redirection symbol (>) sends a log of HELPMAKE activity to the file MY.LOG. You may find it helpful to redirect the log file because, in its more verbose modes (given by /V), HELPMAKE may generate a lengthy log. For additional encoding options, see "Options for Encoding" later in this chapter.

The following command line decodes the Help database MY.HLP into the text file MY.SRC by invoking HELPMAKE with the /O option:

```
HELPMAKE /V /D /O MY.SRC MY.HLP > MY.LOG
```

Once again, the /V option results in verbose output, and the output is directed to the log file MY.LOG. For additional decoding options see "Options for Decoding" later in this chapter.

HELPMAKE Options

HELPMAKE accepts a number of command-line options, which are listed in the sections that follow. You may specify options in uppercase or lowercase letters, and precede them with either a forward slash (/) or a dash (-). For example, -L, /L, -l, and /l all represent the same option.

Most options apply only to encoding; others apply only to decoding; and a few apply to both. See "Options for Encoding" for the options that apply to encoding and "Options for Decoding" for the options that apply to decoding.

Options for Encoding

To encode a database, specify the /E option to HELPMAKE. In addition, you may supply various other options that control the way HELPMAKE encodes the database. All the options that apply when encoding are listed in the following table:

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the Help database file. The character marks a line that contains special information for internal use by the application. For example, the QBX database uses the colon (:).
/C	Indicates that the context strings for this Help file are case sensitive. At run time, all searches for Help topics are case sensitive if the Help database was built with the /C option in effect.

/E[[*n*]]

Creates (encodes) a Help database from a specified text file. The optional *n* indicates the amount of compression to take place. If *n* is omitted, HELPMAKE compresses the file as much as possible, thereby reducing the size of the file by about 50 percent. The more compression requested, the longer HELPMAKE takes to create a database file. The value of *n* is a number in the range 0 – 15. It is the sum of successive powers of 2 representing various compression techniques, as follows:

Value	Technique
0	No compression
1	Run-length compression
2	Keyword compression
4	Extended keyword compression
8	Huffman compression

Add values to combine compression techniques. For example, use /E3 to get run-length and keyword compression. This is useful in the testing stages of Help database creation where you need to create the database quickly and are not too concerned with size.

/H

Displays a summary of HELPMAKE syntax and exits without encoding any files.

/L

Locks the generated file so that it cannot be decoded by HELPMAKE at a later time.

/O*destfile*

Specifies *destfile* as the name of the Help database.

/S*n*

Specifies the type of input file, according to the following /S*n* values:

Option	File type
/S1	Rich Text Format (RTF)
/S2	QuickHelp (default)
/S3	Minimally formatted ASCII

/T

Translates a number of dot commands. Each translated command corresponds to a statement using the application control character, which by default is a colon (:). For example, `.length 20` is equivalent to `:120`. Using /T directs HELPMAKE to perform the following translations:

.category	:c	.next	:>
.command	:x	.mark	:m
.dup	:d	.paste	:p
.end	:e	.popup	:g
.execute	:y	.previous	:<
.file	:f	.ref	:r
.freeze	:z	.suffix	:s
.list	:i	.topic	:n
.length	:l		

/V[[*n*]]

Indicates the amount of diagnostic and informational output, depending on the value of *n*. Increasing the value adds more information to the output. If you omit this option or specify only /V, HELPMAKE gives you its most verbose output. The possible values of /V*n* follow:

Option	Effect
/V	Maximum diagnostic output.
/V0	No diagnostic output and no banner.
/V1	Prints only HELPMAKE banner (default).
/V2	Prints pass names.
/V3	Prints contexts on first pass.
/V4	Prints contexts on each pass.
/V5	Prints any intermediate steps within each pass.
/V6	Prints statistics on Help file and compression.

/W*width*

Indicates the fixed width of the resulting Help text in number of characters. The values of *width* can range from 11 to 255. If the /W option is omitted, the default is 76. When encoding RTF source files (/S1), HELPMAKE automatically formats the text to *width*. When encoding QuickHelp (/S2) or minimally formatted ASCII (/S3) files, HELPMAKE truncates lines to this width.

Options for Decoding

To decode a Help database into QuickHelp files, use the /D option. In addition, HELPMAKE accepts other options to control the decoding process.

The following table shows all the options that are valid when decoding:

Option	Action
/D[[*letter*]]	Decodes the input file into its component parts. If a destination file is not specified with the /O option, the Help file is decoded to stdout. HELPMAKE decodes the file differently depending on the letter specified, as follows:

Option	Effect
/D	Fully decodes the Help database, leaving all cross-references and formatting information intact.
/DS	Decode split. Splits the concatenated, compressed Help database into its components using their original names. If the database was created without concatenation (the default), HELPMAKE simply copies it to a file with its original name. No decompression occurs.
/DU	Decode unformatted. Decompresses the database and removes all screen formatting and cross-references. The output can still be used later for input and recompression, but all screen formatting and cross-references are lost.

/H	Displays a summary of HELPMAKE syntax and exits without decoding any files.
/O[[*destfile*]]	Specifies destination file for the decoded output from HELPMAKE. If *destfile* is omitted, the Help database is decoded to stdout. HELPMAKE always decodes Help database files into QuickHelp format.
/V[[*n*]]	Indicates the amount of diagnostic and informational output displayed depending on the value of *n*. The possible values of *n* follow:

Option	Effect
/V	Maximum diagnostic output.
/V0	No diagnostic output and no banner.
/V1	Prints only the HELPMAKE banner (default).
/V2	Prints pass names.
/V3	Prints contexts on first pass.

Creating a Help Database

You can create a Microsoft-compatible Help database using two methods. The first method is to decompress an existing Help database, modify the resulting Help text file, and recompress the Help text file to form a new database. The second and simpler method is to append a new Help database onto an existing Help database. This method involves the following steps:

1. Create a Help text file in QuickHelp format, RTF, or minimally formatted ASCII.
2. Use HELPMAKE to create a Help database file. The following HELPMAKE command line invokes HELPMAKE, using SAMPLE.TXT as the input file and producing a Help database file named SAMPLE.HLP:

```
HELPMAKE /V /E /OSAMPLE.HLP SAMPLE.TXT > SAMPLE.LOG
```

3. Make a backup copy of the existing database file (for safety's sake).
4. Append the new Help database file onto the existing Help database. The following command line concatenates the new database SAMPLE.HLP onto the end of the BAS7QCK.HLP database using the append feature of the DOS COPY command.

```
COPY BAS7QCK.HLP /B + SAMPLE.HLP /B
```

   You can append additional Help files to BAS7QCK.HLP by adding them to the command line, separating each file with a plus sign (+).

5. Test the database. Suppose that the SAMPLE.HLP database contains the context Sample. If you type the word "Sample" in the QBX environment and request Help on it after appending SAMPLE.HLP to BAS7QCK.HLP, the Help window should display the text associated with the context Sample.

Help Text Conventions

Using common structure and conventions ensures that Help files for one application will make sense when viewed using another. This section outlines organizational conventions used in Help databases supplied by Microsoft. You should follow the same conventions to create Microsoft-compatible Help files.

Context Conventions

Certain contexts are defined by convention across the Help databases for all Microsoft languages. If you decompress any of the Help database files that Microsoft supplies, you will see these contexts in the text output.

Recommended Contexts

The following contexts are recommended, and are present in most Microsoft Help files. If you use these contexts, make sure that they conform to these conventions.

Context	Description
h.contents	The table of contents for the Help file. You can also define the string "contents" for direct reference to this context.
h.index	The index for the Help file. You can also define the string "index" for direct reference to this context.
h.pg#	A specific page within the Help file. This is used in response to a "go to page #" request.

Note that each of the preceding contexts begins with the "h." prefix. The Microsoft Advisor Online Help system considers context strings beginning with *x*., where *x* is a specified character prefix, as internal or constructed Help contexts. Except for the preceding contexts listed, these apply to menu items, error numbers, and so forth; in general, you do not need to insert these in your Help files.

Required Contexts

The following contexts are required and are present in all Microsoft Advisor Help files. If you modify or replace the standard files, be sure to retain these definitions.

Context	Description
h.default	The default Help screen, typically displayed when the user presses Shift+F1 at the top level in most applications. The contents are generally devoted to information on using Help.
h.notfound	The Help text that is displayed when the Help system cannot find information on the requested context. The text could be an index of contexts, a topical list, or general information on using Help.
h.pg1	The Help text that is logically first in the file. This is used by some applications in response to a "go to the beginning" request made within the Help window.
h.pg$	The Help text that is logically last in the file. This is used by some applications in response to a "go to the end" request made within the Help window.

Internal Help Context Prefixes

The following character prefixes denote internal Help contexts:

Character	Description
h.	Help item. Prefixes miscellaneous Help contexts that may be constructed or otherwise hidden from the user. For example, the internal Help context, h.index, refers to the BASIC Language Keyword Index which is part of the QBX online Help.
m.	Menu item. Contexts that relate to product menu items are defined by their accelerator keys. For example, the Exit selection on the file menu item is accessed by Alt+F+X, and is referred to in Help by m.f.x.
e.	Error number. If a product supports the uniform error numbering scheme used by Microsoft languages, it refers to the Help for each error using the error number with the e prefix. For example, the context e.c1234 refers to the C compiler error message number C1234.
d.	Dialogs. Dialogs are assigned a number, and the Help context string is constructed by d. followed by the ASCII text number (for example, d.12).
m.	Messages.

The Help Text File

The Help retrieval facility that is built into Microsoft products is simply a data retrieval tool. It imposes no restrictions on the content and format of the Help text. HELPMAKE and the display routines built into Microsoft language environments, however, make certain assumptions about the format of Help text. This section provides some guidelines for creating Help text files that are compatible with those assumptions.

In all three Help text formats, the Help text source file is a sequence of topics, each preceded by one or more unique context definitions. The following table lists the various formats and the corresponding context-definition statement:

Format	Context definition
QuickHelp	.context *context*
RTF	\par >>*context* \par
Minimally formatted ASCII	>>*context*

In QuickHelp format, each topic begins with one or more .context statements that define the context strings that map to the topic text. Subsequent lines up to the next .context statement constitute the topic text.

In RTF, each context definition must be in a paragraph of its own (denoted by \par), beginning with the Help delimiter (>>). Subsequent paragraphs up to the next context definition constitute the topic text.

In minimally formatted ASCII, each context definition must be on a separate line, and each must begin with the Help delimiter (>>). As in RTF and QuickHelp files, subsequent lines up to the next context definition constitute the topic text.

For detailed information about these three formats see "Using Help Database Formats" later in this chapter.

Explicit Cross-References

Explicit cross-references, or "hyperlinks," in the Help text file are marked with invisible text. A hyperlink comprises a word or phrase followed by invisible text that gives the context to which the hyperlink refers.

When the user activates the hyperlink, the Help system displays the topic named by the invisible text. The keystroke that activates the hyperlink depends on the application. Consult the documentation for each product to find the specific keystroke needed.

To specify hyperlinks in your Help database, you must use either QuickHelp format or RTF. See "Using Help Database Formats" later in this chapter for the specific QuickHelp and RTF options that turns invisible text on and off.

The invisible cross-reference text is formatted as one of the following:

Cross-reference text	Action
context_string	Causes the Help topic associated with *context_string* to be displayed. For example, exe_format results in the display of the Help topic associated with the context exe_format.
filename!	Causes the entire file *filename* to be treated as a single topic to be displayed. For example, $INCLUDE: MYINCL.INC would search the $INCLUDE environment variable for the file MYINCL.INC.
filename!context_string	Works the same way as *context_string*, except that *filename* is searched for the context. If the file is not already open, the Help system finds it (by searching either the current path or an explicit environment variable), and opens it. For example, BASADVR.HLP!FSINR would search the BASADVR.HLP file and bring up the topic associated with FSINR.

Local Contexts

Context strings that begin with an at sign (@) are defined as local and have no implicit cross-references. They are used in cross-references instead of the context string that would otherwise be generated.

When you use a local context, HELPMAKE does not generate a context string that can be used from elsewhere in the Help file. Instead, it embeds a cross-reference that has meaning only within the current Help file. An example of this usage follows:

```
.context normal
This is a normal topic, accessible by the context string "normal."
[button\v@local\v] is a cross-reference to the following topic.
.context @local
```

The topic `@local` can be reached only if the user uses the cross-reference in the previous topic or if the user browses sequentially through the file.

In the example, the text `[button\v@local\v]` refers to `@local` as a local context. If the user selects the text `[button]`, or scrolls through the file, the Help system displays the topic text that follows the context definition for local. Because local is defined with the at sign, it can be accessed only by a hyperlink within the Help file or by sequentially browsing through the file. This saves space in the file and makes it faster to search for and locate the context.

Application-Specific Control Characters

The Help database supports application-specific characters that have special meaning for Microsoft language products. The application-specific character may appear at the beginning of any line of Help text. This special character is interpreted by the application. If the application does not support this character, it is ignored.

Within the databases and applications provided with Microsoft languages, a colon is used as the control character, and the following colon commands are supported:

Command	Action
:l*n*	Indicates the default initial window size, in *n* lines, of the topic about to be displayed. Always the first line in the topic if present.
:n *text*	Defines *text* as the name (or title) to be displayed in place of the context string if the application Help displays a title. Always the first line in the context unless :l is used, in which case :n appears on the line following the :l command.
:p	Indicates a screen break for environment Help. The lines following :p are accessible only by using the PgDn command within the environment-Help dialog box.

Example The following example shows the first two lines from the BASIC Help entry for the **ABS** function:

```
.context ABS
:nABS Function Syntax
   .
   .       (Topic text for ABS)
   .
```

The `:nABS` command directs the application to display the line `HELP: ABS Function Syntax` whenever the context ABS is selected.

Using Help Database Formats

The text format of the database may be any of three types. The following table briefly describes these types. The sections that follow describe each formatting type in detail.

Type	Characteristics
QuickHelp	Uses dot commands and embedded formatting characters (the default formatting type expected by HELPMAKE); supports highlighting, color, and cross-references. This format must be compressed before using.
Minimally formatted ASCII	Uses a Help delimiter (>>) to define Help contexts; does not support highlighting, color, or cross-references. This format may be compressed, but compression is not required.
RTF	Uses a subset of standard RTF; supports highlighting, color, and cross-references. This format must be compressed before using.

An entire Help system (such as the one supplied with Microsoft BASIC) may use any combination of files formatted with different format types. You could, for example, encode a README.DOC information file as minimally formatted ASCII; in the save Help system, you could encode the Help files and user-defined statements and functions in QuickHelp format.

QuickHelp Format

The QuickHelp format uses two dot commands and embedded formatting flags to convey information to HELPMAKE.

The QuickHelp Context Command

To specify a context string, use the .context command. The form of the command is as follows:

.context *context*

One or more .context commands precedes each topic in a QuickHelp file. Each .context command defines a context string for the topic text. You may define more than one context for a single topic, as long as you do not place any topic text between them.

Multiple context commands refer to the first Help-text block, while one context command refers to the second Help-text block. For example, the following shows how the CVI, CVS, CVL, and CVD context strings could all refer to the same topic text, while DATA refers to a separate topic:

```
.context CVI
.context CVS
.context CVL
.context CVD
        .
        . (Topic text describing string-to-number functions)
        .
.context DATA
        .
        . (Topic text describing the DATA statement)
        .
```

The QuickHelp Comment Command

To embed comments in your Help file, use the .comment command. The syntax of this command is as follows:

.comment *text*

Note that this command is useful while you are working in the source file; however, HELPMAKE removes .comment commands and comment text as it encodes the database.

QuickHelp Formatting Flags

The QuickHelp format supports a number of formatting flags that are used to highlight parts of the Help database and to mark hyperlinks in the Help text.

Each formatting flag consists of a backslash (\) followed by a character. The following table lists the formatting flags:

Formatting flag	Action
\\	Inserts a single backslash in text.
\a, \A	Anchors text for cross-references.
\b, \B	Turns boldface on or off.
\i, \I	Turns italics on or off.
\p, \P	Turns off all attributes.
\u, \U	Turns underlining on or off.
\v, \V	Turns invisibility on or off (hides cross-references in text).

On monochrome monitors, text labelled with the bold, italic, and underlining attributes appears in various ways, depending upon the application (for example, high intensity and reverse video are commonly displayed). On color monitors, these attributes are translated by the application into suitable colors, depending on the user's default color selections. Depending on the monitor, \b and \i may produce the same effect. It is a good idea to display a small help file on a target machine as a test before writing a large help file.

The \b, \i, \u, and \v options are toggles, turning on and off their respective attributes. You may use several of these on the same text. Use the \p attribute to turn off all attributes. Use the \v attribute to hide cross-references and hyperlinks in the text.

HELPMAKE truncates the lines in QuickHelp files to the width specified with the /W option. The formatting flags do not count toward the character-width limit. Lines that begin with an application-specific control character are truncated to 255 characters regardless of the width specification. See "Options for Encoding" for more information about the preceding options.

Examples

In the following example, the `\b` flag initiates boldface text for the word CLS and the `\p` flag that follows the word reverts to plain text for the remainder of the line:

```
\bCLS\p - a device I/O statement that clears the display screen.
```

In the following example, the `\a` anchors text for the hyperlink:

```
\aExample\vsample_prog\v
```

The `\v` flags define the cross-reference to be `sample_prog` and causes the text between the flags to be invisible. Cross-references are defined in the following section.

QuickHelp Cross-References

Help databases contain two types of cross-references: implicit and explicit.

An implicit cross-reference is any word that appears both in the topic text and as a context in the Help file. For example, any time you request Help on the word "close," the Help window

will display Help on the **CLOSE** statement. You need not code implicit cross-references in your Help text files.

Explicit cross-references are words or phrases on the screen that are associated with a context. For example, the word "Example" in the initial Help-screen area for a BASIC function is an explicit cross-reference to the program example for that function. You must insert formatting flags in your Help text files to mark explicit cross-references.

If the hyperlink consists of a single word, you can use invisible text to flag it in the source file. The \v formatting flag creates invisible text, as follows:

hyperlink\vcontext\v

Specify the first \v flag immediately following the word you want to use as the hyperlink. Following the flag, insert the context that the hyperlink cross-references. The second \v flag marks the end of the context, that is, the end of the invisible text. HELPMAKE generates a cross-reference whose context is the invisible text, and whose hyperlink is the entire word.

If the hyperlink consists of a phrase, rather than a single word, you must use anchored text to create explicit cross-references. Use the \a and \v flags to create anchored text as follows:

\ahyperlink-words\vcontext\v

The \a flag marks the anchor for the cross-reference. The text that follows the \a flag is the hyperlink. The hyperlink must fit entirely on one line. The first \v flag marks both the end of the hyperlink and the beginning of the invisible text that contains the cross-reference *context*. The second \v flag marks the end of the invisible text.

Examples

In the following example, the topic text for the **SIN** function in the QBX Help database contains three implicit cross-references to the BASIC functions **COS**, **TAN**, and **ATN**:

```
\bSee Also\p  COS\vCOS\v  TAN\vTAN\v  ATN\vATN\v
```

In the following example, the word `Example` is a hyperlink:

```
\bSee also:\p   \uExample\p\vopen.ex\v
```

The hyperlink example refers to `open.ex`. A mouse click or other form of selection with the cursor on any of the letters of `Example` brings up the Help topic whose context is `open.ex`.

```
\bExample\vBASADVR.HLP!.sqrx\v
```

On a color monitor, `Example` appears in boldface and the characters appear highlighted. Any time `Example` is selected, HELPMAKE searches for the context `.sqrx` in the file BASADVR.HLP.

When a hyperlink needs to cross-reference more than one word, you must use an anchor, as shown in the following example:

```
\aProduct Support\v@prodsupp\v
```

Anchored hyperlinks must fit on a single line. In this case, the hyperlink consists of the phrase `Product Support`, which refers to the local context `@prodsupp`. The `\v` flag makes the name `@prodsupp` invisible; it does not appear on the screen when the Help is displayed.

Minimally Formatted ASCII

You can use uncompressed, minimally formatted ASCII Help files instead of compressed QuickHelp format files, although they are larger and slower to search. Minimally formatted ASCII files are of fixed width, may not contain highlighting (or other nondefault attributes) or cross-references, and cannot be larger than 256K.

A minimally formatted ASCII text file comprises a sequence of topics, each preceded by one or more unique context definitions. Each context definition must be on a separate line beginning with a Help delimiter (>>). Subsequent lines up to the next context definition constitute the topic text.

Example The following code shows how Help for a user-defined context would be implemented using minimally formatted ASCII format:

```
>>SAMPLE
SAMPLE Statement Syntax
SAMPLE - A sample statement that performs a user-defined function.
Syntax
  SAMPLE
```

The Help file begins with the Help delimiter (>>), so that HELPMAKE can verify that the file is indeed an ASCII Help file. When displayed, the Help information appears exactly as it is typed into the file. Any formatting codes are treated as ASCII text.

```
>>TROFF
>>TRON
      .
      .   (Topic text describing trace statements)
      .
>>TYPE
      .
      .   (Topic text describing the TYPE statement)
      .
```

To create smaller and faster-to-search minimally formatted ASCII files, compress the files using HELPMAKE.

RTF

Rich Text Format (RTF) is a Microsoft word-processing format supported by many other word processors. It allows documents to be transferred from one application to another without losing any formatting information. HELPMAKE recognizes a subset of the full RTF syntax. If your file contains any RTF code that is not part of the subset, HELPMAKE ignores the code and strips it out of the file.

Certain word-processing and file-conversion programs generate the RTF code automatically as output. You need not worry about inserting RTF codes yourself; you can simply format your Help files directly with a word-processor that generates RTF, using the attributes supported by the subset. The only items you need to insert are the Help delimiter (>>) and context string that start each entry.

HELPMAKE recognizes the subset of RTF listed in the following table:

RTF code	Action
\plain	Default attributes. On most screens this is nonblinking normal intensity.
\b	Boldface. The way this is displayed depends upon the application; often it is intensified text.
\i	Italic. The way this is displayed depends upon the application; often it is reverse video.
\v	Hidden text. Hidden text is used for cross-reference information and for some application-specific communications; it is not displayed.
\ul	Underline. The way this is displayed depends upon the application. For example, if underlining is not supported (either by one application or by the video adapter), it may be displayed as blue text.
\par	End of paragraph.
\pard	Default paragraph formatting.
\fi	Paragraph first-line indent.
\li	Paragraph indent from left margin.
\line	New line (not new paragraph).
\tab	Tab character.

Using the word-processing program, you can break the topic text into paragraphs. When HELPMAKE compresses the file, it formats the text to the width given with the /W option, ignoring the paragraph formats.

As with the other text formats, each entry in the database source consists of one or more context strings, followed by topic text. The Help delimiter (>>) at the beginning of any paragraph denotes the beginning of a new Help entry. The text that follows on the same line is defined as a context for the topic. If the next paragraph also begins with the Help delimiter, it also defines a context string for the same topic text. You may define any number of contexts for a block of topic text. The topic text comprises all subsequent paragraphs up to the next paragraph that begins with the Help delimiter.

Note

RTF uses curly braces for nesting. For a Help database, the contents of the entire file is enclosed in curly braces, as is each specially formatted text item.

Examples

The following code is a portion of the context and topic text for the **BEEP** statement written in RTF format:

```
\par
>>BEEP \par
{\iSyntax}{\bDetails}\vBASADVR.HLP!.beepr\v
{\bBEEP} - a device I/O statement that sounds the speaker
{\bSyntax}
   BEEP
```

The `\i` option displays the word Syntax in reverse video and the `\b` option displays `Details`, `BEEP`, and `Syntax` in bold.

The following code shows two context statements that refer to the same topic text. (Note that the actual RTF output of a word processor normally contains significantly more information that is not visible to the user; this additional information is ignored by HELPMAKE.)

```
{rtf0
\par
>>TROFF \par
>>TRON \par
      .
      .  topic text describing the TROFF and TRON statements
      .
>>TYPE \par
      .
      .  topic text describing the TYPE statement
      .
}
```

HELPMAKE Error Messages

HELPMAKE generates the following error messages:

Number	HELPMAKE Error Message
H1000	/A requires character. The /A option requires an application-specific control character. The correct form is /A*c*, where *c* is the control character.
H1001	/E compression level must be numeric. The /E option requires either no argument or a numeric value. The correct form is /E*n*, where *n* specifies the amount of compression requested.
H1002	Multiple /O parameters specified. Only one output file can be specified with the /O option.
H1003	Invalid /S filetype identifier. The /S option requires specification of the type of input file. There was an invalid file-type identifier specified. The correct form is /S*n*, where *n* specifies the format of the input Help-text file. The only valid values are 1 (RTF), 2 (QuickHelp format), and 3 (minimally formatted ASCII).
H1004	/S requires filetype identifier. The /S option requires specification of the type of input file. There was no file-type identifier specified. The correct form is /S*n*, where *n* specifies the format of the input Help-text file. The only valid values are 1 (RTF), 2 (QuickHelp format), and 3 (minimally formatted ASCII).
H1005	/W fixed width invalid. The /W option requires a width specification. The width specification was either not specified or outside valid range (11-255).
H1050	Improper arguments for /DS. The /O, /L, and /C options are invalid with the /DS option.
H1051	Improper arguments for /D. The /D option permits only no argument, an S, or a U. In addition, it may not be used with /L or /C.

H1052 Encode requires /O option.

You have requested database encoding without specifying an output-file name for the operation.

H1097 No operation specified.

There is no operation specified on the HELPMAKE command line. Either the /D or the /E option must be specified.

H1098 Unknown Switch.

There is an invalid option specified on the command line.

H1099 Syntax error on command line.

HELPMAKE cannot interpret the command line.

H1100 Cannot open file.

One of the files specified on the HELPMAKE command line could not be found or created.

H1101 Error writing file.

The output file could not be written, probably because the disk is full.

H1102 No Input File Specified.

In an encoding operation, no input Help-text file was specified.

H1103 No context strings found.

No context strings were found in the input stream while encoding. Either the file is empty, or the specified /S value does not correspond to the Help text formatting.

H1104 No topic text found.

No topic text was found in the Help text file. Either the file is empty, or the specified /S value does not correspond to the Help text format.

H1200 Insufficient memory to allocate context buffer.

There is insufficient memory to run HELPMAKE. It requires 256K free memory.

H1201 Insufficient memory to allocate utility buffer.

There is insufficient memory to run HELPMAKE. It requires 256K free memory.

H1250 Not a valid compressed Help file.

The input file specified for a decompression operation is not a valid Help database file.

H1251 Cannot decompress locked Help file.

The Help database file you are attempting to decompress is locked (that is, the /L option was specified when the Help file was created).

H1300 Word too long in RTF processing.

A single word was longer than the specified format width (set by /W option).

H1301 Too much back-up required while formatting RTF.

While attempting to reformat a paragraph, HELPMAKE had to back up more than 128 characters to find a word break.

H1302 Attribute stack overflow processing RTF.

RTF attributes are nested too deeply. HELPMAKE supports a maximum of 50 levels of attribute nesting.

H1303 Unknown RTF attribute.

An unknown RTF formatting command was encountered. HELPMAKE needs to understand all RTF formatting commands; either an unrecognized RTF command was encountered, or the wrong /S*n* option was specified

H1900 Internal Virtual Memory Error.

This message indicates an internal HELPMAKE error. Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form included in your documentation.

H1901 Out of local memory.

This message indicates an internal HELPMAKE error. Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form included in your documentation

H1902 Out of disk space for swap file.

HELPMAKE uses a temporary swapping file which is written to the current drive and directory. That drive is full. The temporary file may grow to 1.5 times the size of the input files (for large Help files) and is not removed until the final Help file is completed.

H1903 Cannot open swapping file.

HELPMAKE uses a temporary swapping file, which is written to the current drive and directory. It cannot create the swapping file because the disk drive or directory is full.

H1990 Character not found in cmpr.

This message indicates an internal HELPMAKE error. Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form included in your documentation.

H2000 Line too long, truncated.

A line exceeded the fixed width specified by the /W option. The extra characters have been truncated.

H2001 Duplicate context string.

The same context string appeared preceding more than one block of topic text. A context string may be associated with one and only one block of topic text.

H4000 Keyword compression analysis table size exceeded.

This error occurs in conjunction with HELPMAKE error H4001. The maximum number (16,000) of unique keywords has been encountered during keyword compression. This happens only in very large Help files. No further keywords will be included in the analysis.

H4001 No further new words will be analyzed.

This error occurs in conjunction with HELPMAKE error H4000. There is no more room for keywords in the analysis tables. HELPMAKE continues to analyze how frequently words that it has already encountered occur.

Appendixes

Appendix A Language Elements 695

Appendix B Data Types, Constants, Variables and Arrays 701

Appendix C Operators and Expressions 731

Appendix D Programs and Modules 741

Appendix A
Language Elements

As in virtually all programming languages, characters from the BASIC character set form labels, keywords, variables, and operators. These are combined to form the statements that make up a program.

This appendix discusses the following topics:

- The BASIC character set and the special meanings of some characters
- Program line syntax
- Line identifiers
- Executable and nonexecutable statements
- Program line length

Appendix B, "Data Types, Constants, Variables, and Arrays" and Appendix C, "Operators and Expressions," discuss other parts of statements, including expressions.

Character Set

The Microsoft BASIC character set consists of alphabetic characters, numeric characters, and special characters.

The alphabetic characters in BASIC are the uppercase letters (A–Z) and lowercase letters (a–z).

The BASIC numeric characters are the digits 0–9. The letters A–F and a–f can be used as parts of hexadecimal numbers.

Table A.1 lists the special characters and their meanings.

Table A.1 Special Characters

Character	Name
Enter	Terminates input of a line
	Blank (or space)
!	Exclamation point (suffix for single-precision data type)
#	Number sign (suffix for double-precision data type)
$	Dollar sign (suffix for string data type)
%	Percent (suffix for integer data type)
&	Ampersand (suffix for long-integer data type)

Table A.1 *Continued*

Character	Name
@	At sign (suffix for currency data type)
'	Single quotation mark (apostrophe)
(	Left parenthesis
)	Right parenthesis
*	Asterisk (multiplication symbol)
+	Plus sign
,	Comma
–	Minus sign
.	Period (decimal point)
/	Slash (division symbol)
:	Colon
;	Semicolon
<	Less than
=	Equal sign (relational operator or assignment symbol)
>	Greater than
?	Question mark
[	Left bracket
]	Right bracket
\	Backslash (integer division symbol)
^	Up arrow or caret (exponentiation symbol)
_	Underscore (line continuation symbol)

The BASIC Program Line

BASIC program lines have the following syntax:

[[*line-identifier*]] [[*statement*]] [[: *statement*]]... [[*comment*]]

Line Identifiers

BASIC supports two types of *line-identifiers*: line numbers and alphanumeric line labels.

Line Numbers

A line number may be any combination of digits from 0 to 65,529. The following are examples of valid line numbers:

```
1
200
300PRINT "hello"     '300 is the line number.
65000
```

Using 0 as a line number is not recommended. Error-trapping and event-trapping statements (such as **ON ERROR** and **ON** *event*) interpret the presence of line number 0 to mean that trapping is disabled (stopped). For example, the following statement disables error trapping and does not branch to line 0 if an error occurs:

```
ON ERROR GOTO 0
```

In addition, **RESUME 0** continues execution on the line where the error occurred, not at line number 0.

Alphanumeric Line Labels

An alphanumeric line label may be any combination of 1 to 40 letters and digits, starting with a letter and ending with a colon. BASIC keywords are not permitted. The following are valid alphanumeric line labels:

```
Alpha:
ScreenSub:
Test3A:
```

Case is not significant; the following line labels are equivalent:

```
alpha:
Alpha:
ALPHA:
```

Line numbers and labels may begin in any column, as long as they are the first characters other than blanks or tabs on the line. Blanks and tabs are also allowed between an alphanumeric label and the colon following it. A line can have only one label.

BASIC does not require each line in a source program to have the same type of identifier, either a line number or label. You may mix alphanumeric labels and line numbers in the same program, and you may use alphanumeric labels as objects of any BASIC statement where line numbers are permitted, except as the object of an **IF...THEN** statement. In **IF...THEN** statements, BASIC permits only a line number, unless you explicitly use a **GOTO** statement. For example, the form of the following statement is correct:

```
IF A = 10 THEN 500
```

However, if the object of the **IF...THEN** statement is a line label, a **GOTO** statement is required:

```
IF A = 10 THEN GOTO IncomeData
```

If you are trapping errors, the **ERL** function returns only the last line number located before the error. Neither **RESUME** nor **RESUME NEXT** requires line labels or line numbers. See the entries for **ERL** and **RESUME** in the *BASIC Language Reference* for more information.

Line numbers do not determine the order in which statements are executed. For example, BASIC executes statements in the following program in the order 100, 10, 5:

```
100 PRINT "The first line executed."
 10 PRINT "The second line executed."
  5 PRINT "The third and final line executed."
```

Some older BASICs, such as BASICA, would execute the lines in numerical order: 5, 10, 100.

Executable and Nonexecutable Statements

A BASIC statement is either "executable" or "nonexecutable." An executable statement advances the flow of a program's logic by telling the program what to do next (telling it to read input, write output, add two numbers together and store the resulting value in a variable, open a file, branch to another part of the program, or take some other action). In contrast, a nonexecutable statement does not advance the flow of a program's logic. Instead, nonexecutable statements perform tasks such as allocating storage for variables, declaring and defining variable types, and designating variables to be shared among all the procedures in a source file.

The following statements are nonexecutable:

COMMON	**DIM** (static arrays only)
CONST	**OPTION BASE**
DATA	**SHARED**
DECLARE	**STATIC**
DEF*type*	**TYPE...END TYPE**

Another type of nonexecutable statement is a "comment" used to clarify a program's operation and purpose. A comment is introduced by the **REM** statement or a single quote character (**'**). The following lines are equivalent:

```
PRINT "Quantity remaining" : REM Print report label.
PRINT "Quantity remaining"   ' Print report label.
```

More than one statement can be placed on a line, but colons (:) must separate statements, as illustrated by the following:

```
FOR I=1 TO 5 : PRINT "G'day, mate." : NEXT I
```

Line Length

If you create your programs using QBX, you are limited to lines of 256 characters. QBX does not recognize the underscore character (_) as a line continuation. When QBX loads your program, the underscores are removed and the continued lines are joined to form a single line.

If you use your own editor then compile your programs from the compiler command line, you may use an underscore as the last character to create a program line that extends across more than one physical line:

```
IF (TestChar$ = " " OR TestChar$ = ".") AND _
LineNumber < 23 AND NOT EOF(FileNumber) THEN
```

The line-length limit is not imposed in this case.

Underscores cannot be used to continue **DATA** or **REM** statements, even when you compile from the command line.

Appendix B
Data Types, Constants, Variables, and Arrays

This appendix covers the following topics:

- Data types
- Constants
- Variables
- Scope rules for variables and constants
- Static and dynamic arrays
- Automatic and static variables in **SUB** and **FUNCTION** procedures
- Numeric type conversions

Data Types

Every variable in BASIC has a data type that determines what can be stored in the variable. There are two categories of data in BASIC: string data and numeric data. Each category includes elementary data types.

Elementary Data Types —String

The two kinds of strings are variable-length strings and fixed-length strings.

Variable-Length Strings

A variable-length string is can contain up to 32,767 characters. The codes for these characters range from 0–127 for ASCII characters, and from 128–255 for non-ASCII characters (see Appendix A, "Keyboard Scan and ASCII Character Codes" in the *BASIC Language Reference*).

Fixed-length Strings

A fixed-length string contains a declared number of characters. Fixed-length strings can be no longer than 32,767 characters. Fixed-length strings, like variable-length strings, contain characters with codes ranging from 0–255.

Elementary Data Types—Numeric

BASIC has two integer types, two floating-point types, and a currency type. Table B.1 summarizes the types.

Table B.1 Numeric Data Types

Numeric Type	Suffix	Description
Integer (2 bytes)	%	Integers are stored as signed 16-bit binary numbers ranging in value from –32,768 to 32,767.
Long integer (4 bytes)	&	Long integers are stored as signed 32-bit binary numbers ranging in value from –2,147,483,648 to 2,147,483,647.
Single-precision floating point (4 bytes)	!	Single-precision numbers are accurate to seven decimal places, and have ranges of –3.402823E38 to –1.40129E–45, zero, and 1.40129E–45 to 3.402823E38. When using the alternate math library (compiling with /Fpa), the range of values is –3.402823E38 to –1.175494E–38, zero, and 1.175494E–38 to 3.402823E38.
Double-precision floating point (8 bytes)	#	Double-precision numbers are accurate to 15 to 16 digits and have ranges of –1.7976931348623 15D308 to –4.94065D–324, zero, and 4.94065D–324 to 1.7976931348623 15D308. When using the alternate math library (compiling with /Fpa), the range of values is –1.79769313486232D308 to –2.2250738585072D–308, zero, and 2.2250738585072D–308 to 1.79769313486232D308.
Currency	@	Currency numbers are fixed-point numbers stored as signed 8-byte integers scaled by 10,000. They are accurate to 19 digits and have a range of –922337203685477.5808 to 922337203685477.5807.

Integer Numbers

All BASIC integers are represented as two's complement values. Integers use 16 bits (2 bytes) and long integers use 32 bits (4 bytes).

In two's complement representation, positive values are represented as straightforward binary numbers. For example, BASIC would store an integer value of 4 as a sequence of 16 bits:

```
0000000000000100
```

Negative values are represented as the two's complement of the corresponding positive value. To form the two's complement (the negative) of the integer value 4, first take the representation above and complement all bits (change all 1s to 0s and all 0s to 1s):

```
1111111111111011
```

Then add one to the result:

```
1111111111111100
```

The final result is how BASIC represents – 4 as a binary number.

Because of the way two's complement numbers are formed, every combination of bits representing a negative value has a 1 as the leftmost bit.

Floating-Point Numbers

BASIC uses IEEE-format floating-point numbers. IEEE format gives very accurate results and makes it possible to use a math coprocessor.

Floating-point values are represented in a different format than integers. In floating-point format each number consists of three parts: the sign, the exponent, and the mantissa. You can think of this format as a variation of scientific notation. In scientific notation, the number 1000 would be represented as 1.0×10^3. To save space, you could just remember the exponent 3 and the mantissa 1.0 and reconstruct the value later by raising 10 to the power 3 and multiplying by the mantissa. Floating-point notation works by saving just the exponent and the mantissa. The only difference is that in this format the exponent represents a power of 2, not a power of 10.

In a single-precision number, the sign takes 1 bit, the exponent takes 8 bits, and the mantissa uses the remaining 23 bits and an additional implied bit. Double-precision values occupy 8 bytes or 64 bits: 1 bit for the sign, 11 bits for the exponent, 52 actual bits for the mantissa and an implied bit.

The following program (included on the BASIC distribution disks in the file FLPT.BAS) can be used to examine the internal format of single-precision values.

```
DECLARE FUNCTION MHex$ (X AS INTEGER)
' Display how a given real value is stored in memory.

DEFINT A-Z
DIM Bytes(3)
CLS
PRINT "Internal format of IEEE number (in hexadecimal)"
PRINT
DO

' Get the value.
INPUT "Enter the value (END to end): ", A$
IF UCASE$(A$) = "END" THEN EXIT DO
RealValue! = VAL(A$)
' Convert the real value to a long without changing any of
' the bits.
AsLong& = CVL(MKS$(RealValue!))
' Make a string of hex digits, and add leading zeros.
Strout$ = HEX$(AsLong&)
```

```
Strout$ = STRING$(8 - LEN(Strout$), "0") + Strout$
' Save the sign bit, and then eliminate it so it doesn't
' affect breaking out the bytes.
SignBit&=AsLong& AND &H80000000&
AsLong&=AsLong& AND &H7FFFFFFF&
' Split the real value into four separate bytes.
' --the AND removes unwanted bits; dividing by 256 shifts
' the value right 8 bit positions.
FOR I = 0 TO 3
   Bytes(I) = AsLong& AND &HFF&
   AsLong& = AsLong& \ 256&
NEXT I
' Display how the value appears in memory.
PRINT
PRINT "Bytes in Memory"
PRINT "  High        Low"
FOR I = 1 TO 7 STEP 2
   PRINT "  "; MID$(Strout$, I, 2);
NEXT I
PRINT:PRINT
' Set the value displayed for the sign bit.
Sign = ABS(SignBit& <> 0)

' The exponent is the right 7 bits of byte 3 and the
' leftmost bit of byte 2. Multiplying by 2 shifts left and
' makes room for the additional bit from byte 2.
Exponent = Bytes(3) * 2 + Bytes(2) \ 128
' The first part of the mantissa is the right 7 bits
' of byte 2. The OR operation makes sure the implied bit
' is displayed by setting the leftmost bit.
Mant1 = (Bytes(2) OR &H80)
PRINT " Bit 31    Bits 30-23  Implied Bit & Bits 22-0"
PRINT "Sign Bit  Exponent Bits     Mantissa Bits"
PRINT TAB(4); Sign; TAB(17); MHex$(Exponent);
PRINT TAB(33); MHex$(Mant1);MHex$(Bytes(1));MHex$(Bytes(0))
PRINT
LOOP

' MHex$ makes sure we always get two hex digits.
FUNCTION MHex$ (X AS INTEGER) STATIC
   D$ = HEX$(X)
   IF LEN(D$) < 2 THEN D$ = "0" + D$
   MHex$ = D$
END FUNCTION
```

Output

```
Enter the value (END to end): 4

Bytes in Memory
  High     Low
  40 80 00 00

 Bit 31     Bits 30-23  Implied Bit & Bits 22-0
Sign Bit   Exponent Bits      Mantissa Bits
    0            81               800000
```

The program displays the sign, exponent, and mantissa of the single-precision value. The mantissa's implied bit is automatically included. All values are printed out in hexadecimal values (base 16 numbers), which are a shorthand way of writing binary numbers. Each hexadecimal digit represents a pattern of 4 bits: &H0 (0_{HEX}) represents 0000 binary, &H8 (8_{HEX}) represents 1000, and &HF (F_{HEX}) represents 1111.

Looking at the output from the program, the decimal value 4 is represented in memory as a series of 4 bytes:

```
40 80 00 00
```

These 4 bytes break down into a single-sign bit of 0, a 1-byte exponent of &H81, and a mantissa of &H800000.

The exponent value of &H81 represents a biased exponent, not the true exponent. With a biased exponent, a fixed value (a "bias") is added to the true exponent and the result is stored as part of the number. For single-precision values, the bias is &H7F or 127 decimal. Double-precision values use a bias of &H3FF or 1023 decimal.

Using a biased exponent avoids having to use one of the exponent bits to represent the sign. In the output, 4 is a power of 2, so the true exponent is 2. Adding the bias (&H7F) gives the stored exponent value of &H81.

A normalized mantissa refers to a mantissa that is multiplied by 2 (shifted to the left) and the exponent decreased until the leftmost bit is a 1. By eliminating leading 0s from the mantissa, more significant bits can be stored. In memory, the mantissa for a value of 4 is all 0s. This is because the mantissa is normalized and the leftmost bit is assumed.

Because the mantissa is always normalized, the leftmost bit is always a 1. And, because the leftmost bit is always a 1, there is no reason to store it as part of the number. BASIC therefore stores 23 bits (bit 22 to bit 0) for the mantissa of a single-precision number but also includes a 24th implied bit that is always a 1.

There is an implied "binary point" (analogous to a decimal point) to the right of the implied bit. The binary point indicates that bits 22 to 0 in the mantissa are really a binary fraction. Thus, in the example, the mantissa is really a single 1 (the implied bit) followed by a zero binary fraction (bits 22 to 0). The mantissa of 1 multiplied by 2 taken to the power of the exponent 2 is $1 * 2^2$, or 4.

Currency Numbers

A currency number is stored as an 8-byte two's-complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation gives a range of:

$$+\left(\frac{2^{63}-1}{10000}\right) = +922337203685477.5807$$

to

$$-\left(\frac{2^{63}}{10000}\right) = -922337203685477.5808$$

The currency data type is extremely useful for calculations involving money, or for any fixed-point calculation where high speed is more important than limited fixed-point fraction precision. Because currency numbers are stored as integers, BASIC uses integer routines for **ABS**, **SGN**, **FIX**, **INT**, – (negation), +, and – (subtraction).

User-Defined Data Types

BASIC lets you define new data types using the **TYPE** statement. A user-defined type is an aggregate type made up of elementary types. For example, the following **TYPE** statement defines a type, `InventoryItem`:

```
TYPE InventoryItem
   Quantity              AS LONG
   OrderPoint            AS LONG
   Cost                  AS CURRENCY
   VendorCodes(1 TO 10)AS INTEGER
   Description           AS STRING*25
   Number                AS STRING*10
END TYPE
```

The new type `InventoryItem` contains two long integers, one currency item, a static array of integers, and two fixed-length strings. A variable of a user-defined type occupies only as much storage as the sum of its components. `InventoryItem` takes up 71 bytes: 4 bytes each for the two long integers (8 bytes total), 8 bytes for the currency item, 20 bytes for the static array, and 35 bytes for the fixed-length strings.

You may use any of the BASIC data types except variable-length strings in a user-defined type: short and long integers, single- and double-precision floating-point values, currency, fixed-length strings, static arrays, and other user-defined types. However, user-defined types cannot include dynamic arrays.

Data Types in ISAM Files

Some special restrictions apply to BASIC ISAM files. When you create an ISAM table, you create a user-defined data type by including an appropriate **TYPE...END TYPE** statement in the declarations part of your program. Table B.2 lists the data types you can specify in a **TYPE...END TYPE** statement.

Table B.2 ISAM Data Types

Data type	Size limit	Description	Indexable
INTEGER	2 bytes, signed	Whole numbers in the range –32,768 to 32,767.	Yes
LONG	4 bytes, signed	Whole numbers in the range –2,147,483,648 to 2,147,483,647.	Yes
DOUBLE	8 bytes	Real numbers in the following ranges:– 1.797693134862315D 308 to –4.94065D –324, zero, and 4.94065D–324 to 1.797693134862315D308.	Yes
CURRENCY	8 bytes	Use to store fixed point (such as dollar and cents) values in the approximate range ±9.22E14 with accuracy to 19 digits.	Yes
STRING	Up to 32K	Use to store string data.	Only if shorter than 256 bytes
Static Array	64K	Raw binary	No
User-defined type (structure)	64K	Raw binary	No

Note BASIC's **SINGLE** data type is not supported in ISAM; use **DOUBLE** or **CURRENCY** instead.

Constants

Constants are predefined values that do not change during program execution. There are two general types of constants: literal and symbolic. Literal constants may be either string constants or numeric constants.

String Constants

A string constant can contain up to 32,767 characters. These characters can be any of the characters (except the double quote character and any carriage-return and line-feed sequences) whose ASCII codes fall within the range 0–255. This range includes the actual ASCII characters (0–127) and the extended characters (128–255). The following are all valid string constants:

```
HELLO
$25,000.000
Number of Employees
```

Numeric Constants

Numeric constants are positive or negative numbers. Table B.3 describes numeric constants and provides examples.

Table B.3 Types of Numeric Constants

Type	Subtype	Description	Examples

Integer	Decimal	One or more decimal digits (0–9) with an optional sign prefix (+ or –). The range for integer decimal constants is –32,768 to 32,767.	`68`
			`+407`
			`-1`

Integer	Hexadecimal	One or more hexadecimal digits (0–9, a–f, or A–F) with the prefix &H or &h. The range for integer hexadecimal constants is &h0 to &hFFFF.	`&H76` `&H32F`
Integer	Octal	One or more octal digits (0–7) with the prefix &O, &o, or &. The range for integer octal constants is &o0 to &o177777.	`&o347` `&1234`

Table B.3 *Continued*

Type	Subtype	Description	Examples
Long integer	Decimal	One or more decimal digits (0–9) with an optional sign prefix (+ or –) and the suffix &. The range for long decimal constants is –2,147,483,648 to 2,147,483,647.	`95000000` `-400141`
Long integer	Hexadecimal	One or more hexadecimal digits (0–9, a–f, or A–F) with the prefix &H or &h and the suffix &. The range for long hexadecimal constants is &h0& to &hFFFFFFFF&.	`&H0&` `&H1AAAAA&`
Long integer	Octal	One or more octal digits (0–7) with the prefix &O, &o, or & and the suffix &. The range for long octal constants is &o0& to &o37777777777&.	`&o347&` `&555577733&`
Fixed point		Positive or negative real numbers—numbers containing decimal points.	`9.0846`
Currency		Positive or negative real numbers—numbers containing decimal points, with 19 digits of precision, and having at most 4 digits to the right of the decimal point. The range for currency constants is ±9.22E14	`39.9965`

Table B.3 *Continued*

Type	Subtype	Description	Examples
Floating point	Single precision	Positive or negative numbers represented in exponential form (similar to scientific notation). A single-precision floating-point constant is an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The constant's value is the mantissa multiplied by the power of ten represented by the exponent. The range for single-precision constants is –3.40E38 to 3.40E38.	`2235.988E-7` `2359E6`
Floating point	Double precision	Double-precision floating-point constants have the same form as single-precision floating-point constants, but use D, rather than E, to indicate the exponent. Double-precision constants have a range of –1.79D308 to 1.79D308.	`4.35D-10`

Single-Precision Constants

Single-precision numeric constants are stored with 7 digits of precision (plus the exponent). A single-precision constant is any numeric constant that has one of the following properties:

- Exponential form denoted by "E."
- A trailing exclamation mark (!).
- A value containing a decimal point that does not have either a "D" in the exponent or a trailing number sign (#), and that has fewer than 8 digits.
- A value without a decimal point that has fewer than 15 digits but cannot be represented as a long-integer value.

Example The following are examples of single-precision constants:

```
46.8
-1.09E-6
3489.0
22!
```

Note Numeric constants in BASIC cannot contain commas.

Double-Precision Constants

Double-precision numbers are stored with 15 digits of precision (plus the exponent). A double-precision constant is any numeric constant that has one of the following properties:

- Exponential form denoted by "D."
- A trailing number sign (#).
- A decimal point, no "E" in the exponent or trailing exclamation mark (!), and more than 7 digits.

Example The following are examples of double-precision constants:

```
345692811.5375901
-1.09432D-06
3489.0#
987654321.1234567
```

Currency Constants

Currency constants are fixed-point numbers stored with 19 digits of precision and have, at most, 4 digits to the right of the decimal point. Currency constants have the following properties:

- Fixed-point form with no more than 4 digits to the right of the decimal point, within the currency data type range, and a trailing at symbol (@).
- Integer form (no decimal point) within the currency data type range, and a trailing at symbol (@).

Symbolic Constants

BASIC provides symbolic constants that can be used in place of numeric or string values. The following fragment declares two symbolic constants and uses one to dimension an array:

```
CONST MAXCHARS%=254, MAXBUF%=MAXCHARS%+1
DIM Buffer%(MAXBUF%)
```

The name of a symbolic constant follows the same rules as a BASIC variable name. You may include a type-declaration character (**%**, **&**, **!**, **#**, **@**, and **$**) in the name to indicate its type, but this character is not part of the name. For example, after the following declaration, the names `N!`, `N#`, `N$`, `N%`, `N@`, and `N&` cannot be used as variable names because they have the same name as the constant:

```
CONST N=45
```

A constant's type is determined either by an explicit type-declaration character or the type of the expression. Symbolic constants are unaffected by **DEF***type* statements.

If you omit the type-declaration character, the constant is given a type based on the expression. Strings always yield a string constant. With numeric expressions, the expression is evaluated and the constant is given the simplest type that can represent it. For example, if the expression gives a result that can be represented as an integer, the constant is given an integer type.

See the section, "Scope of Variables and Constants" later in this appendix for information about the scope of constants. See also the entry for **CONST** in the *BASIC Language Reference* for more information about where and how to use symbolic constants.

Variables

A variable is a name that refers to an object—a particular number, string, or record. (A record is a variable declared to be a user-defined type.) Simple variables refer to a single number, string, or record. Array variables refer to a group of objects, all of the same type.

A numeric variable, whether simple or array, can be assigned only a numeric value (either integer, long integer, single precision, or double precision). A string variable can be assigned only a character-string value. You can assign one record variable to another only if both variables are the same user-defined type. However, you can always assign individual elements of a record to a variable of the corresponding type.

The following list shows some examples of variable assignments.

Variable assignment	Example
A constant value	`A = 4.5`

The value of another string or numeric variable	`B$ = "ship of fools"`
	`A$ = B$`
	`Profits = NetEarnings`

The value of a record element	`TYPE EmployeeRec`
	`  EName AS STRING*25`
	`  SocSec AS STRING*9`
	`END TYPE`
	`DIM CurrentEmp AS EmployeeRec`
	`.`
	`.`
	`.`
	`OutSocSec$=CurrentEmp.SocSec`

The value of one record variable to another of the same type	`TYPE FileBuffer`
	`  EName AS STRING*25`
	`  JobClass AS INTEGER`
	`END TYPE`
	`DIM Buffer1 AS FileBuffer`
	`DIM Buffer2 AS FileBuffer`
	`.`
	`.`
	`.`
	`Buffer2=Buffer1`

The value obtained by combining other variables, constants, and operators	`CONST PI = 3.141593`
	`Conversion = 180/PI`
	`TempFile$ = FileSpec$+".BAK"`

Note Before a variable is assigned a value, its value is assumed to be zero (for numeric variables) or null (for string variables). All fields in a record, including string fields, are initialized to zero. See Appendix C, "Operators and Expressions," for more information on the operators used in BASIC for combining variables and constants.

Variable Names

A BASIC variable name may contain up to 40 characters. The characters allowed in a variable name are letters, numbers, the decimal point, and the type-declaration characters (**%**, **&**, **!**, **#**, **@**, and **$**).

The first character in a variable name must be a letter. If a variable begins with **FN**, it is assumed to be a call to a **DEF FN** function. (As a general rule, the use of a **FUNCTION** is preferred over a **DEF FN**.)

A variable name cannot be a reserved word, but embedded reserved words are allowed. For example, the following statement is illegal because **LOG** is a reserved word (BASIC is not case sensitive):

```
Log = 8
```

However, the following statement is legal:

```
TimeLog = 8
```

Reserved words include all BASIC commands, statements, function names, and operator names (see Appendix B, "BASIC Reserved Words," in the *BASIC Language Reference* for a complete list of reserved words).

Variable names must also be distinct from both **SUB** and **FUNCTION** procedure names and symbolic constant (**CONST**) names.

ISAM Names

Some parts of an ISAM database require names (for example, tables, columns, and indexes), and these names must conform to the ISAM naming convention. The ISAM convention is essentially a subset of the BASIC convention, as shown in the following table.

BASIC naming convention	ISAM naming convention
40 characters or fewer.	30 characters or fewer
Alphanumeric characters, plus the BASIC type-declaration characters, where appropriate (variables and functions).	Alphanumeric characters only, including A–Z, a–z, and 0–9
Must begin with alphabetic character, but only DEF FN functions can begin with "fn."	Must begin with alphabetic character
The period is not allowed in the names of elements within a user-defined type. Because these are the names BASIC and ISAM have in common, there is no conflict.	No special characters allowed
Not case sensitive.	Not case sensitive

Declaring Variable Types

Simple variables can be numeric, string, or record variables. You may specify simple variable types by the use of a type-declaration suffix, in an **AS** declaration statement, or in a **DEF***type* declaration statement. Variables may also be declared as arrays.

Type-Declaration Suffix

BASIC uses the following type-declaration suffixes:

% & ! # @ $

The dollar sign ($) is the type-declaration character for string variables; that is, it "declares" that the variable represents a string. The following is an example of a string declaration:

```
A$ = "SALES REPORT"
```

Numeric variable names can declare integer values (denoted by the % suffix), long-integer values (denoted by the & suffix), single-precision values (denoted by the ! suffix), double-precision values (denoted by the # suffix), or currency values (denoted by the @ suffix). Single precision is the default for variables without a suffix.

There is no type-declaration character for a user-defined type.

AS Declaration Statements

You can declare variables using the following syntax:

declare variablename **AS** *type*

In the preceding syntax, *declare* can be either **DIM, COMMON, SHARED,** or **STATIC,** and *type* can be either **INTEGER, LONG, SINGLE, DOUBLE, CURRENCY, STRING,** or a user-defined type. For example, the following statement declares the variable `A` as having a long-integer type:

```
DIM A AS LONG
```

String variables declared in an **AS STRING** clause can be either variable-length strings or fixed-length strings. Variable-length strings are expandable—their length depends on the length of any string assigned to them. Fixed-length strings have a constant length, specified by adding **number* to the **AS STRING** clause, where *number* is the length of the string in bytes.

Example

The following example shows how variable- and fixed-length strings handle the same data:

```
' String1 can have a variable length:
DIM String1 AS STRING

' String2 has a fixed length of 7 bytes:
DIM String2 AS STRING*7
```

```
String1  = "1234567890"
String2  = "1234567890"
PRINT String1
PRINT String2
```

Output

```
1234567890
1234567
```

For more information on fixed-length and variable-length strings, see Chapters 4, "String Processing," 11, "Advanced String Storage," and 13, "Mixed-Language Programming with Far Strings."

You can declare record variables by using the name of the user type in the **AS** clause:

```
TYPE InventoryItem
   Quantity              AS LONG
   OrderPoint            AS LONG
   Cost                  AS CURRENCY
   VendorCodes(1 TO 10)AS INTEGER
   Description           AS STRING*25
   Number                AS STRING*10
END TYPE
DIM CurrentItem AS InventoryItem, PreviousItem AS InventoryItem
```

Use the format *variablename.elementname* to refer to individual elements of the new variable, as in the following example:

```
IF CurrentItem.Description = "Ergonomic Desk Chair" THEN
   PRINT CurrentItem.Number; CurrentItem.Quantity
END IF
```

If you declare a variable with an **AS** clause, every declaration of the variable must use the **AS** clause. For example, in the following fragment the **AS** clause is required in the **COMMON** statement because **AS** was also used in the **DIM** statement:

```
CONST MAXEMPLOYEES=250
DIM EmpNames(MAXEMPLOYEES) AS STRING
COMMON EmpNames() AS STRING
.
.
.
```

DEFtype Declaration Statements

Use the BASIC statements **DEFINT, DEFLNG, DEFSTR, DEFSNG, DEFDBL,** and **DEFCUR** to declare the types for certain variable names. By using one of these **DEF***type* statements, you can specify that all variables starting with a given letter or range of letters are one of the elementary variable types, without using the trailing declaration character.

DEF*type* statements only affect variable names in the module in which they appear. See the **DEF***type* entry in the *BASIC Language Reference* for more information.

The type-declaration suffixes for variable names, the type names accepted in **AS** *type* declarations, and the memory (in bytes) required for each variable type to store the variable's value are listed in Table B.4.

Table B.4 Variable-Type Memory Requirements

Suffix	AS *type* name	Variable type	Size of data
%	**INTEGER**	Integer	2
&	**LONG**	Long integer	4
!	**SINGLE**	Single precision	4
#	**DOUBLE**	Double precision	8
@	**CURRENCY**	Scaled integer	8
$	**STRING**	Variable-length string	Takes 4 bytes for descriptor, 1 byte for each character in string
$	**STRING***num*	Fixed-length string	Takes *num* bytes
None	Declared user-defined type	Record variable	Takes as many bytes as the individual elements require

Declaring Array Variables

An array is a group of objects referenced with the same variable name. The individual values in an array are elements. Array elements are also variables and can be used in any BASIC statement or function that uses variables. You dimension an array when you use it the first time or when you declare the name, type, and number of elements in the array.

Each element in an array is referred to by an array variable subscripted with an integer or an integer expression. You may use noninteger numeric expressions as array subscripts; however, they are rounded to integer values. The name of an array variable has as many subscripts as there are dimensions in the array. For example, `V(10)` refers to a value in a one-dimensional array, while `T$(1,4)` refers to a value in a two-dimensional string array. The default upper subscript value for any array dimension is 10. The maximum subscript value and the number of dimensions can be set by using the **DIM** statement. The maximum number of dimensions for an array is 60. The maximum number of elements per dimension is 32,767. See the **DIM** entry in the *BASIC Language Reference* for more information.

You may have arrays of any simple variable type, including records. To declare an array of records, first declare the data type in a **TYPE** statement and then dimension the array:

```
TYPE TreeNode
   LeftPtr AS INTEGER
   RightPtr AS INTEGER
   DataField AS STRING*20
END TYPE
DIM Tree(500) AS TreeNode
```

Each element of the array `Tree` is a record of type `TreeNode`. To use a particular element of a record in an array, use the dot notation form *variablename.elementname*:

```
CONST MAXEMPLOYEES=500
TYPE EmployeeRec
   EName  AS STRING*25
   SocSec AS STRING*9
END TYPE
DIM Employees(MAXEMPLOYEES) AS EmployeeRec
.
.
.
PRINT Employees(I).EName;" ";Employees(I).SocSec
```

Array names are distinct from simple variable names. The array variable `T` and the simple variable `T` in the following example are two different variables:

```
DIM T(11)
T = 2 : T(0) = 1          'T is simple variable.
FOR I% = 0 TO 10          'T(0) is element of array.
   T(I% + 1) = T * T(I%)
NEXT
```

Array elements, like simple variables, require a certain amount of memory, depending on the variable type. See Table B.4 for information on the memory requirements for storing individual array elements.

To find the total amount of memory required by an array, multiply the number of elements by the bytes per element required for the array type. For example, consider the following two arrays:

```
DIM Array1(1 TO 100) AS INTEGER
DIM Array#(-5 TO 5)
```

The first array, `Array1`, has 100 integer elements, so its values take 200 bytes of memory. The second array, `Array2`, has 11 double-precision elements, so its values require 88 bytes of memory. Because BASIC must store information about the array along with the array's values, arrays take more memory than just the space for the values.

Variable Storage and Memory Use

Under the DOS and OS/2 operating systems, RAM stores the resident portion of any BASIC program that is currently running, the various constants and data needed by the program, variables, and any other information needed by the computer while the program is running.

RAM used by a BASIC program is divided into two categories: near memory and far memory. Near memory and far memory each contain a "heap" which is an area of memory used to store dynamic variables. "Near memory," also referred to as "DGROUP," is the single segment of memory (maximum size of 64K) that includes, but is not limited to, the near heap (where near strings and variables are stored), the stack, and state information about the BASIC run-time. "Far memory" is the multiple-segment area of memory outside of DGROUP that includes, but is not limited to, the BASIC program (run-time and generated code) and the far heap (where dynamic arrays and far strings are stored).

Variable-Length String Storage

You have two options for storing variable-length string data and string array data: near string storage in DGROUP and far string storage in the far heap. You can specify which option you want to use by compiling with or without the far string option (/Fs). The only data type affected by the /Fs option is the variable-length string data type. Table B.5 shows where variables and arrays are stored in memory based on the storage option you choose.

Table B.5 Storage of Data in BASIC

Type of BASIC data	Compiler options	Memory location
Simple numeric[1] variables	All	DGROUP
Variable-length string descriptors	All	DGROUP
Variable-length string data	Far Strings (/Fs)	Far Heap
Near strings	Default	DGROUP
All numeric[1] array descriptors	All	DGROUP
Static numeric[1] array data	All	DGROUP
Dynamic numeric[1] array data	All	Far Heap
Huge dynamic numeric[1] array data	Huge Arrays (/Ah)	Far Heap

[1] Denotes **INTEGER, LONG, SINGLE, DOUBLE, CURRENCY**, and user-defined data types and fixed-length strings.

Note When you compile from within QBX, the default compile option is far string (/Fs) if a Quick library is loaded, and near string otherwise. When you compile from the command line, near string storage is the default, as it was in all previous versions of BASIC. To use far string storage instead, add the /Fs option to the BASIC Compiler (BC) command line.

Because the QBX environment treats all strings as far strings, a program that uses near pointers to string data cannot run or be debugged in the QBX environment. However, the program may still work when compiled using the near string compiler option, and can be debugged with CodeView. On the other hand, far pointers will always work in QBX and in a program compiled with far strings. For detailed information about string storage, see Chapter 11, "Advanced String Storage."

String Array Storage

A string array has three parts in memory: the array descriptor, the array of string descriptors, and the string data. The array descriptor is always stored in DGROUP. Each element in the array of string descriptors is stored in DGROUP and contains the length and location in memory of the string data. The string data resides in the near heap if near string storage is specified at compile time, or in far heap if far string storage is specified at compile time. The 4-byte string descriptor for each variable-length string resides in DGROUP regardless of which string option (near or far) is used.

BASIC string arrays can be either static or dynamic. A "static string array" is an array of variable length strings whose descriptors reside in a permanently allocated array in DGROUP. This array of descriptors is fixed when compiled, and cannot be altered while a program is running.

A "dynamic string array" is a string array whose array of descriptors can be defined or changed during run time with BASIC's **DIM**, **REDIM**, or **ERASE** statements. Dynamic arrays that are declared local to a procedure are de-allocated when control leaves the procedure. As with static string arrays, dynamic string array descriptors also reside in DGROUP, but they may change in number and/or location during execution of a BASIC program.

The location of the string data itself is independent of the static or dynamic status of a string array and depends solely on whether the near or far string option is chosen when compiling.

Numeric Array Storage

Unlike string arrays, numeric arrays have a fixed amount of data associated with each element. Therefore, the following information about numeric arrays is equally applicable to arrays of fixed-length strings or arrays of user-defined types, since they also have a fixed amount of data associated with each element in the array.

A numeric array has two parts: an array descriptor and the numeric array data itself. The array descriptor contains information about the array including its type, dimensions, and the location of its data in memory. Array descriptors are always stored in DGROUP. The data in a numeric array may be stored entirely in DGROUP or entirely in the far heap.

Huge Dynamic Array Storage

Huge dynamic arrays allow you to create and store in memory arrays that contain more than 64K of data (64K is the limitation of standard dynamic arrays). Huge arrays are invoked by using the /Ah option when starting QBX or by using the /Ah option when compiling a program from the command line.

Scope of Variables and Constants

Any time a variable appears, BASIC follows a set of rules in determining to which object the variable refers. These rules describe a variable's scope—the range of statements over which the variable is defined.

The BASICA interpreter has a very simple scope rule: a variable exists when you first use it and persists until the program ends. The scope of a variable is from its first use through the end of the program.

In BASIC, you can control the scope of variables and symbolic constants—which helps you write compact, well-defined **SUB** and **FUNCTION** procedures that don't interfere with each other. You can also make some variables available to all procedures in a module, and thereby share important data structures among procedures.

You may think of variables and constants as having one of two scopes: global or local. Global variables, once declared, may be used anywhere in a module to refer to some single object. Local variables are local to some part of the module, the module-level code, or one of the procedures. In addition, variables can be shared in such a way that they aren't quite global, nor are they completely local.

The rest of this section discusses global and local scope, and shared variables. The following skeleton of a program is used in this discussion. The program, a main program and two procedures, replaces runs of blanks in a file with tab characters—a simple first step in compressing a file. (The program is included on the BASIC distribution disks in the file ENTAB.BAS.)

```
DEFINT a-z
DECLARE FUNCTION ThisIsATab(Column AS INTEGER)

CONST MAXLINE=255, TABSPACE=8
CONST NO=0, YES=NOT NO
DIM SHARED TabStops(MAXLINE)
.
.
.
' Set the tab positions (uses the global array TabStops).
CALL SetTabPos
.
.
.
```

```
   IF ThisIsATab(CurrentColumn) THEN
          PRINT CHR$(9);
          LastColumn=CurrentColumn
   END IF
.
.
.

'==================SUB SetTabPos=========================
' Set the tab positions in the array TabStops.
'
SUB SetTabPos STATIC
   FOR I=1 TO MAXLINE
          TabStops(I)=((I MOD TABSPACE)=1)
   NEXT I
END SUB

'===============FUNCTION ThisIsATab=====================
' Answer the question, "Is this a tab position?"
'
FUNCTION ThisIsATab(LastColumn AS INTEGER) STATIC
   IF LastColumn>MAXLINE THEN
          ThisIsATab=YES
   ELSE
          ThisIsATab=TabStops(LastColumn)
   END IF
END FUNCTION
```

Global Variables and Constants

Variables and symbolic constants can be global in BASIC programs. A global variable or global symbolic constant is defined for the entire module. For a variable, the only way to make it global is to declare it in the module-level code with the **SHARED** attribute in a **DIM**, **REDIM**, or **COMMON** statement. A symbolic constant is a global constant if it is declared in the module-level code using a **CONST** statement.

In the sample program, the array `TabStops` is a global variable. Because `TabStops` is declared in a **DIM** statement with the **SHARED** attribute, it is shared with every procedure in the module. Notice that both procedures in the program (`ThisIsATab` and `SetTabPos`) use `TabStops` by making a reference to it. Global variables do not require any additional declarations to be used in procedures in the module. Similarly, the symbolic constants `MAXLINE` and `TABSPACE` are global constants. If you use the name of a global variable or constant in a procedure, you are referring to the global variable and not a local variable of the same name.

Note The **SHARED** statement allows procedures to share variables with the module-level code. This is not the same as making the variable global. See the section "Sharing Variables" later in this appendix for more information.

Local Variables and Constants

A local variable or constant exists only within a procedure or the module-level code. If the name of a local variable is used in another procedure in a module, the name represents a different variable and refers to a different object.

The sample program ENTAB.BAS, described previously, includes many local variables. The variables `CurrentColumn` and `LastColumn` are local to the module-level code. The variable `I` is local to the subprogram `SetTabPos`. Finally, the variable `LastColumn` in the parameter list of `ThisIsATab` can be thought of as a local variable because it is a formal parameter. (Remember, however, that a formal parameter stands for the actual argument passed to the procedure. See the section "Passing Arguments to Procedures" in Appendix D, "Modules and Procedures," for additional information.)

It is simplest to think of a local variable as any variable that isn't global. Any variable that appears in module-level code or in a procedure is local if it isn't declared in a **DIM**, **REDIM**, or **COMMON** statement with the **SHARED** attribute. (There is one exception. See the following section, "Sharing Variables.") Even if a variable appears in one of these statements, you may still use a local variable of the same name in a procedure by declaring the variable in a **STATIC** statement. If the sample program had a **DIM SHARED** statement declaring `I` to be a global variable in the main program, you could make `I` a local variable in `SetTabPos` by adding a **STATIC** statement just after the **SUB** statement:

```
SUB SetTabPos STATIC
STATIC I
.
.
.
```

Any symbolic constant declared inside a **SUB** or **FUNCTION** procedure is a local constant. For example, in the following fragment, `ENDOFLIST` is a local symbolic constant that exists only in the function `FindElement`:

```
FUNCTION FindElement(X())
CONST ENDOFLIST = -32767
.
.
.
END FUNCTION
```

Note The STATIC statement not only declares a variable to be local: it also directs the compiler to save the value of the variable between procedure calls. Do not use **STATIC** statements in recursive procedures if you do not want a variable's value saved between calls. See the section "Automatic and Static Variables" later in this appendix for more information.

Sharing Variables

You can share variables among parts of a module without making the variables global by using the **SHARED** statement. For example, to share `TabStops` without making it a global variable, you would remove the **SHARED** attribute in the module-level code and add **SHARED** statements to the two procedures:

```
DIM TabStops(MAXLINE)
.
.
.
SUB SetTabPos STATIC
SHARED TabStops()
.
.
.
FUNCTION ThisIsATab(LastColumn AS INTEGER) STATIC
SHARED TabStops()
.
.
.
```

The **SHARED** statements indicate that the name `TabStops` in both procedures refers to the same variable defined at the module level.

DEF FN Functions

The **DEF FN** function is an exception to the BASIC scope rules. Every variable in a **DEF FN** function that isn't in its parameter list is part of the module-level code. To make a variable local to a **DEF FN**, you must declare the variable in a **STATIC** statement.

The **STATIC** statement in the following **DEF FN** function makes the variable `I` local:

```
CONST NO = 0, YES = NOT NO

DEF FNIsThereAZ (A$)
STATIC I
   FOR I = 1 TO LEN(A$)
          IF UCASE$(MID$(A$, I, 1)) = "Z" THEN
                FNIsThereAZ = YES
                EXIT DEF
          END IF
   NEXT I
   FNIsThereAZ = NO
END DEF
```

Remember, as a general rule, a **FUNCTION** is preferred over a **DEF FN** because of the improved portability and increased modularity the **FUNCTION** provides.

Summary of Scope Rules

The following list summarizes BASIC's scope rules:

- A variable declared in a **DIM**, **REDIM**, or **COMMON** statement with the **SHARED** attribute is a global variable. Any **SUB** or **FUNCTION** procedure can refer to it.
- A symbolic constant is global if it is declared in a **CONST** statement in the module-level code. Symbolic constants declared in a **SUB** or **FUNCTION** are local.
- A variable is a local variable if it appears in a procedure and is not declared as a global variable. You can use the name of a global variable as a local variable in a procedure by declaring it in the procedure with the **STATIC** statement or by using it as a formal parameter.
- The **SHARED** statement lets you share a variable with the module-level code and other procedures with equivalent **SHARED** statements without making the variable a global variable.
- All variables in a **DEF FN** function are part of the module-level code unless they are either explicitly made local in a **STATIC** statement or are formal parameters.

Static and Dynamic Arrays

You can get better control of your program's use of memory by controlling when storage is set aside for arrays. Storage for arrays can be set aside when the program is compiled or when the program is running. Arrays given storage when the program is compiled are static arrays. Dynamic arrays have storage set aside when the program is running. The storage taken by dynamic arrays can be eliminated while the program is running in order to free memory for other uses. See the entries for **DIM**, **ERASE**, and **REDIM** in the *BASIC Language Reference* for specific information about manipulating dynamic arrays.

How an array is declared can determine whether the array is static or dynamic. By default, arrays dimensioned with constant subscripts or arrays that are implicitly dimensioned are static arrays. Arrays dimensioned with variable subscripts or that are first declared in a **COMMON** statement are dynamic arrays. In a **SUB** or **FUNCTION** not declared static, all arrays are dynamic.

You can also use the **$STATIC** and **$DYNAMIC** metacommands to control how array storage is allocated. However, the **$STATIC** metacommand cannot force arrays to be static in a procedure not declared static; in such a procedure all arrays are dynamic. See the *BASIC Language Reference* for more information.

In some cases, you can allow more space for strings by replacing static arrays with dynamic arrays, or by using variable-length string arrays stored in far memory. In programs run as executable files, the space for static arrays is allocated from DGROUP, an area where strings are stored. On the other hand, dynamic arrays and variable-length string arrays in far memory do not take any space in DGROUP; they are stored as far objects and require far addresses.

Automatic and Static Variables

BASIC procedures can use automatic and static variables. Automatic variables are initialized at the start of each call to the **FUNCTION** or **SUB**; static variables retain values between calls.

You can control whether the default is automatic or static by using or omitting the **STATIC** keyword in the **SUB** or **FUNCTION** statement. If you omit **STATIC**, then the default for variables is automatic. When you use **STATIC**, the default for all variables in the procedure is static—the values of the variables are saved between procedure calls.

You can make selected variables in a procedure static by making the default automatic (omitting **STATIC** from the **SUB** or **FUNCTION** statement) and using the **STATIC** statement.

Example The following program uses a **FUNCTION** that has two static variables: `Start%` and `SaveStr$`. The other variables are automatic. The **FUNCTION** takes a string and returns one token—a string of characters—until the end of the string is reached. On the first call, `StrTok$` makes a local copy of the string `Srce$` in the static variable `SaveStr$`. After the first call, `StrTok$` returns additional tokens from the string using the static variable `Start%` to remember where it left off. All of the other variables (`BegPos%`, `Ln%`, etc.) are automatic. (This program is included on the BASIC distribution disks under the filename TOKEN.BAS.)

```
DECLARE FUNCTION StrTok$(Source$,Delimiters$)

LINE INPUT "Enter string: ",P$
' Set up the characters that separate tokens.
Delimiters$=" ,;:().?"+CHR$(9)+CHR$(34)
' Invoke StrTok$ with the string to tokenize.
Token$=StrTok$(P$,Delimiters$)
WHILE Token$<>""
   PRINT Token$
   ' Call StrTok$ with a null string so it knows this
   ' isn't the first call.
   Token$=StrTok$("",Delimiters$)
WEND

FUNCTION StrTok$(Srce$,Delim$)
STATIC Start%, SaveStr$

   ' If first call, make a copy of the string.
   IF Srce$<>"" THEN
         Start%=1 : SaveStr$=Srce$
   END IF

   BegPos%=Start% : Ln%=LEN(SaveStr$)
   ' Look for start of a token (character that isn't a delimiter).
   WHILE(BegPos%<=Ln% AND INSTR(Delim$,MID$(SaveStr$,BegPos%,1))<>0)
         BegPos%=BegPos%+1
   WEND
   ' Test for token start found.
   IF BegPos% > Ln% THEN
         StrTok$="" : EXIT FUNCTION
   END IF
   ' Find the end of the token.
   EndTok%=BegPos%
   WHILE(EndTok%<=Ln%AND INSTR(Delim$,MID$(SaveStr$,EndTok%,1))=0)
         EndTok%=EndTok%+1
   WEND
   StrTok$=MID$(SaveStr$,BegPos%,EndTok%-BegPos%)
      ' Set starting point for search for next token.
      Start%=EndTok%

END FUNCTION
```

Output

```
Enter string: Allen spoke: "What's a hacker, anyway?"
Allen
spoke
What's
a
hacker
anyway
```

Type Conversion

When necessary, BASIC converts a numeric constant from one type to another, according to the rules defined in the following sections.

Numeric Constants and Numeric Variables

If a numeric constant of one type is set equal to a numeric variable of a different type, the numeric constant is stored as the type declared in the variable name.

Example The following example shows this:

```
A% = 23.42
PRINT A%
```

Output

```
23
```

Strings and Numerics

If a string variable is set equal to a numeric value, or vice versa, an error message is generated that reads `Type Mismatch`.

Type Conversion in Expression Evaluation

During expression evaluation, the operands in an arithmetic or relational operation are converted to the same degree of precision, that of the most precise operand, as each operation is performed. Also, the result of an arithmetic operation is returned to the final degree of precision.

Examples The following example shows how BASIC handles different levels of precision in an arithmetic operation:

```
X% = 2 : Y! = 1.5 : Z# = 100
A! = X% / Y!
PRINT A! * Z#
```

Output

```
133.3333373069763
```

Although the preceding output is displayed in double precision (because of the double-precision variable `Z#`), it has only single-precision accuracy because the assignment to `A!` forced the result of `X% / Y!` to be reduced to single-precision accuracy. This explains the nonsignificant digits (`73069763`) after the fifth decimal place. Contrast this with the output from the following example in which the intermediate result of `X% / Y!` is retained in double-precision.

```
X% = 2 : Y# = 1.5 : Z# = 100
PRINT X% / Y# * Z#
```

Output

```
133.3333333333333
```

Type Conversion in Logical Expressions

Logical operators such as **AND** and **NOT** convert their operands to long integers if necessary. Operands must be in the range –2,147,483,648 to 2,147,483,647 or an `Overflow` error message is generated. See Appendix C, "Operators and Expressions," for more information on logical operators.

Type Conversion Between Floating Point and Integer Values

When a floating-point value is converted to an integer, the fractional portion is rounded.

Example The following example shows this.

```
Total% = 55.88
PRINT Total%
```

Output

```
56
```

Appendix C
Operators and Expressions

This appendix discusses how to combine, modify, compare, or get information about expressions by using the five kinds of operators available in BASIC.

Operators and Expressions Defined

Anytime you do a calculation or manipulate a string, you are using expressions and operators. An expression can be a string or numeric constant, a variable, or a single value obtained by combining constants, variables, and other expressions with operators.

Operators perform mathematical or logical operations on values. There are five categories of operators in BASIC:

- Arithmetic
- Relational
- Logical
- Functional
- String

Hierarchy of Operations

The BASIC operators have an order of precedence. When several operations take place within the same program statement, operations are executed in the following order:

1. Arithmetic operations
   a. Exponentiation (^)
   b. Negation (–)
   c. Multiplication and division (*, /)
   d. Integer division (\)
   e. Modulo arithmetic (**MOD**)
   f. Addition and subtraction (+,–)
2. Relational operations (=, >, <, <>, <=, >=)

3. Logical operations
   a. **NOT**
   b. **AND**
   c. **OR**
   d. **XOR**
   e. **EQV**
   f. **IMP**

An exception to the preceding order of operations occurs when an expression has adjacent exponentiation and negation operators. In this case, the negation is done first. For example, the following statement prints the value .0625 (equivalent to 4^{-2}), not –16 (equivalent to $-(4^2)$):

```
PRINT 4 ^ - 2
```

If the operations are different and are of the same level, the leftmost operation is executed first and the rightmost last:

```
A = 3 + 6 / 12 * 3 - 2     'A = 2.5
```

The order of operations in the preceding example is as follows:

Operation	Result
6 / 12	0.5
0.5 * 3	1.5
3 + 1.5	4.5
4.5 – 2	2.5

In a series of additions or a series of multiplications, there is no fixed evaluation order. Either `3 + 5` or `5 + 6` may be calculated first in the following statement:

```
C = 3 + 5 + 6
```

Usually this does not cause problems. However, it may cause a problem if you have a series of **FUNCTION** procedure calls:

```
C = Incr(X) + Decr(X) + F(X)
```

If any of the three **FUNCTION** procedures modify `X` or change shared variables, the result depends on the order in which BASIC does the additions. You can avoid this situation by assigning the results of the **FUNCTION** calls to temporary variables and then performing the addition:

```
T1 = Incr(X) : T2 = Decr(X) : T3 = F(X)
C = T1 + T2 + T3
```

Arithmetic Operators

Parentheses change the order in which arithmetic operations are performed. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained. The following table lists sample algebraic expressions and their BASIC counterparts.

Algebraic expression	BASIC expression
$\frac{X-Y}{Z}$	`(X-Y) /Z`
$\frac{XY}{Z}$	`X*Y/Z`
$\frac{X+Y}{Z}$	`(X+Y) /Z`
$(X^2)^Y$	`(X^2) ^Y`
X^{YZ}	`X^ (Y*Z)`
$X(-Y)$	`X* (-Y)`

Generally, you must separate two consecutive operators with parentheses. Exceptions to this rule are `* -`, `*  +`, `^ -`, and `^ +`. You could also write `X* (-Y)` as `X*-Y`.

See the preceding section for information about order of arithmetic operations.

Integer Division

Integer division is denoted by the backslash (\) instead of the forward slash (/) (which indicates floating-point division). Before integer division is performed, operands are rounded to integers or long integers, so they must be greater than –2,147,483,648.5 and less than 2,147,483,647.5. The quotient of an integer division is truncated to an integer.

Example The following example contrasts integer and floating-point division:

```
PRINT 10\4, 10/4, -32768.499\10, -32768.499/10
```

Output

```
2             2.5           -3276         -3276.8499
```

Modulo Arithmetic

Modulo arithmetic is denoted by the modulus operator, **MOD**. Modulo arithmetic provides the remainder, rather than the quotient, of an integer division.

Example The following example shows a modulo operation:

```
X% = 10.4\4
REMAINDER% = INT(10.4) - 4*X% ' 10\4 = 2, with remainder 2
PRINT REMAINDER%, 10.4 MOD 4
```

Output

```
2               2
```

Overflow and Division by Zero

Dividing by zero, raising zero to a negative power, and arithmetic overflow produce run-time errors. These errors can be trapped by an error-trapping routine. See Chapter 8, "Error Handling," for more information about error trapping.

Relational Operators

You can use relational operators to compare two values. The result of the comparison is either true (nonzero) or false (zero). You can use the result to make a decision regarding program flow. Although BASIC treats any nonzero value as true, true is usually represented by –1.

Table C.1 lists the relational operators.

Table C.1 Relational Operators and Their Functions

Operator	Relation tested	Expression
=	Equality[1]	X = Y
<>	Inequality	X <> Y
<	Less than	X < Y
>	Greater than	X > Y
<=	Less than or equal to	X <= Y
>=	Greater than or equal to	X >= Y

1 The equal sign is also used to assign a value to a variable.

If you combine arithmetic and relational operators in one expression, the arithmetic operations are always done first. For example, the following expression is true if the value of `X + Y` is less than the value of `(T - 1)/Z`:

```
X + Y < (T - 1)/Z
```

Be careful using relational operators with single- and double-precision values. Calculations may give extremely close but not identical results. In particular, avoid testing for identity between two values. For example, the **PRINT** statement in the following **IF** statement is not executed unless `A!` is exactly equal to 0.0:

```
IF A! = 0.0 THEN PRINT "Exact result."
```

When `A!` is an extremely small value, for example 1.0E–23, the **PRINT** statement is not executed. In addition, a compiled program (.EXE file) may give different results than the same program run in the QBX environment.

Files with the .EXE extension contain more efficient code that may change the way single- and double-precision values are compared. For example, the following fragment prints `Equal` when run in the QBX environment, but prints `Not Equal` when compiled:

```
B!=1.0
A!=B!/3.0
IF A!=B!/3.0 THEN PRINT "Equal" ELSE PRINT "Not Equal"
```

Because the compiled version makes more extensive use of a math coprocessor chip (or coprocessor emulation), `A!` and `B!/3.0` are slightly different values.

You can avoid problems in comparisons by performing calculations outside comparisons. The following rewritten fragment produces the same results whether you run it under QBX or compile it:

```
B!=1.0
A!=B!/3.0
Tmp!=B!/3.0
IF A!=Tmp! THEN PRINT "Equal" ELSE PRINT "Not Equal"
```

Logical Operators

Logical operators perform tests on multiple relations, bit manipulations, or Boolean operations. They return a true (nonzero) or false (zero) value to be used in making a decision.

```
IF D < 200 AND F < 4 THEN 80

WHILE I > 10 OR K < 0
.
.
.
WEND

IF NOT P THEN PRINT "Name not found"
```

The six BASIC logical operators are listed in Table C.2 in order of precedence.

Table C.2 BASIC Logical Operators

Operator	Meaning
NOT	Bit-wise complement
AND	Conjunction
OR	Disjunction (inclusive "or")
XOR	Exclusive "or"
EQV	Equivalence
IMP	Implication

Each operator returns results as indicated in Table C.3. "T" indicates a true value and "F" indicates a false value. Operators are listed in order of operator precedence.

Table C.3 Values Returned by Logical Operations

Values of		Value returned by logical operator					
X	Y	NOT X	X AND Y	X OR Y	X XOR Y	X EQV Y	X IMP Y
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	F	T
F	F	T	F	F	F	T	T

In an expression, logical operations (also known as Boolean operations) are performed after arithmetic and relational operations. Operands are converted to integers (or, if necessary, long integers) before the logical operation is done. The operands of logical operators must be in the range -2,147,483,648 to 2,147,483,647. If the operands are not in this range, an error results. If the operands are either 0 or –1, logical operators return 0 or –1 as the result, as in the following example.

Example The following example prints a "truth table":

```
PRINT " X        Y       NOT      AND      OR       ";
PRINT "XOR      EQV      IMP"
PRINT
I = 10 : J = 15
X = (I = 10) : Y = (J = 15)   'X is true (-1); Y is true (-1)
CALL TruthTable (X,Y)
X = (I > 9) : Y = (J > 15)    'X is true (-1); Y is false (0)
CALL TruthTable (X,Y)
X = (I <> 10) : Y = (J < 16) 'X is false (0); Y is true (-1)
CALL TruthTable (X,Y)
X = (I < 10) : Y = (J < 15)   'X is false (0); Y is false (0)
CALL TruthTable (X,Y)
END

SUB TruthTable(X,Y) STATIC
 PRINT X "     " Y "     ";NOT X "   " X AND Y "      " X OR Y;
 PRINT "     " X XOR Y "     " X EQV Y "     " X IMP Y
 PRINT
END SUB
```

Output

```
X       Y      NOT     AND      OR      XOR     EQV     IMP

-1      -1       0      -1      -1       0      -1      -1

-1       0       0       0      -1      -1       0       0

 0      -1      -1       0      -1      -1       0      -1

 0       0      -1       0       0       0      -1      -1
```

Note the similarity of the output from this program to Table C.3: "T" becomes –1 and "F" becomes 0. Logical operators compare each bit of the first operand with the corresponding bit in the second operand to compute the bit in the result. In these bit-wise comparisons, a 0 bit is equivalent to a false value (F) in Table C.3, while a 1 bit is equivalent to a true value (T).

It is possible to use logical operators to test bytes for a particular bit pattern. For example, the **AND** operator can be used to mask all but one of the bits of a status byte, while the **OR** operator can be used to merge two bytes to create a particular binary value.

```
PRINT 63 AND 16
PRINT -1 AND 8
PRINT 10 OR 9
PRINT 10 XOR 10,                ' Always 0
PRINT NOT 10, NOT 11, NOT 0     ' NOT X = -(X + 1)
```

Output

```
16
 8
 11
 0     -11     -12     -1
```

The first **PRINT** statement uses **AND** to combine 63 (111111 binary) and 16 (10000). When BASIC calculates the result of an **AND**, it combines the numbers bit by bit, producing a 1 only when both bits are 1. Because the only bit that is a 1 in both numbers is the fifth bit, only the fifth bit in the result is a 1. The result is 16, or 10000 in binary. In the second **PRINT** statement, the numbers –1 (binary 1111111111111111) and 8 (binary 1000) are combined using another **AND** operation. The only bit that is a 1 in both numbers is the fourth bit, so the result is 8 decimal or 1000 binary. The third **PRINT** statement uses an **OR** to combine 10 (binary 1010) and 9 (binary 1001). An **OR** produces a 1 bit whenever either bit is a 1, so the result of the **OR** in the third **PRINT** is 11 (binary 1011). The **XOR** in the fourth **PRINT** statement combines the number 10 (1010 binary) with itself. The result is a 0 because an **XOR** produces a 1 only when either, but not both, bits are 1.

Performing a **NOT** on a number changes all 1s to 0s and all 0s to 1s. Because of the way two's complement numbers work, taking the **NOT** of a value is the same as adding one to the number and then negating the number. In the final **PRINT** statement, the expression `NOT 10` yields a result of –11.

Functional Operators

You can use a function in an expression to call a predetermined operation to be performed on an operand. For example, **SQR** is a functional operator used twice in the following assignment statement:

```
A = SQR (20.25) + SQR (37)
```

BASIC incorporates two kinds of functions: intrinsic and user-defined. Many predefined (intrinsic) functions are built into the language. Examples are the **SQR** (square root) and **SIN** (sine) functions.

You can define your own functions with the **FUNCTION...END FUNCTION** construction and the older, obsolete **DEF FN...END DEF** construction. Such functions are defined only for the life of the program (unless they are in a Quick library) and are not part of the BASIC language. In addition to **FUNCTION** and **DEF FN**, you can define subprograms with **SUB**. For more information on defining your own functions and subprograms, see Appendix D, "Modules and Procedures," and Chapter 2, "SUB and FUNCTION Procedures." Also, see the *BASIC Language Reference*.

String Operators

A string expression consists of string constants, string variables, and other string expressions combined by string operators. There are two classes of string operations: concatenation and string function.

The act of combining two strings is called concatenation. The plus symbol (+) is the concatenation operator for strings.

Example The following example program fragment combines the string variables `A$` and `B$` to produce the output shown.

```
A$ = "FILE": B$ = "NAME"
PRINT A$ + B$
PRINT "NEW " + A$ + B$
```

Output

```
FILENAME
NEW FILENAME
```

You can compare strings using the following relational operators (see Table C.1):

`<` `>` `=` `<>` `<=` `>=`

Note that these are the same relational operators used with numbers.

String comparisons are made by taking corresponding characters from each string and comparing their ASCII codes. If the ASCII codes are the same for all the characters in both strings, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If the end of one string is reached during string comparison, the shorter string is smaller if they are equal up to that point. Leading and trailing blanks are significant. The following are examples of true string expressions:

```
"AA" < "AB"
"FILENAME" = "FILE"+"NAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"                    'where B$ = "8/12/85"
```

You can use string comparisons to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks. For more information about ASCII codes, see Appendix A, "Keyboard Scan Codes and ASCII Character Codes," in the *BASIC Language Reference*.

Appendix D
Programs and Modules

This appendix describes how modules are organized and how BASIC procedures communicate with other parts of a program.

Specifically, this appendix covers the following topics:

- Module organization
- The different kinds of procedures
- Passing arguments to procedures
- Recursion
- Overlays

Modules

BASIC programs consist of one or more modules. A module is a source file that can be separately compiled. Declarations, executable statements—any BASIC statement—can appear in a module.

A module may contain **SUB** and **FUNCTION** procedures, as well as code not directly part of a **SUB** or **FUNCTION**. Statements that are not part of a **SUB** or **FUNCTION** are called module-level code. Module-level code includes declarative statements like **DIM** and **TYPE**, as well as error- and event-handling code.

A program has one special module called the main module. The main module contains the entry point of the program (the place where the program starts running). Each program contains only one main module, and the module-level code it contains corresponds to what is often called the main program.

Procedures

This section discusses two BASIC procedures: **FUNCTION** procedures and **SUB** procedures. It also covers the less-powerful **DEF FN** procedure and compares it with the other procedures.

See Chapter 2, "SUB and FUNCTION Procedures," for examples and information about when to use different kinds of procedures.

FUNCTION Procedures

FUNCTION procedures provide a powerful alternative to **DEF FN** procedures. Like **DEF FN** procedures, **FUNCTION** procedures are used in expressions and directly return a single value. There are, however, important differences.

FUNCTION procedures pass values by reference, so a **FUNCTION** procedure can return additional values by changing variables in its argument list. In addition, **FUNCTION** procedures can be used recursively—a function can call itself (see the section "Recursion" later in this appendix for more information).

Unlike **DEF FN** procedures, a **FUNCTION** procedure may be used outside the module in which it is defined. You must include a **DECLARE** statement if you use a **FUNCTION** defined in another module. BASIC automatically generates **DECLARE** statements for **FUNCTION** procedures defined and used in the same module. You can also enter the **DECLARE** yourself:

```
DECLARE FUNCTION Log10(X)

INPUT "Enter a number: ",Num
PRINT "10 ^ Log10(";Num;") is" 10.0^Log10(Num)
END
' Function to find log base 10 of a number using
' BASIC's built-in natural logarithm function.
FUNCTION Log10 (X) STATIC
   Log10=LOG(X)/LOG(10.0)
END FUNCTION
```

FUNCTION procedures also differ from **DEF FN** procedures in that they are not part of the module-level code.

SUB Procedures

Unlike **DEF FN** and **FUNCTION** procedures, a **SUB** procedure is invoked as a separate statement:

```
' Print a message in the middle of the screen.
CLS
CALL PrntMsg(12,40,"Hello!")
END

' Print message at the designated row and column.
SUB PrntMsg(Row%,Col%,Message$) STATIC
   CurRow%=CSRLIN                        ' Save current cursor position.
   CurCol%=POS(0)
   LOCATE Row%,Col% : PRINT Message$;' Print the message at the location.
   LOCATE CurRow%,CurCol%                ' Restore cursor location.
   END SUB
```

SUB procedures can be used to return multiple values to a calling routine and are not invoked as part of an expression.

All **SUB** arguments are passed by reference. This allows **SUB** procedures to return values by changing variables in the argument list—the only way a **SUB** can return a value.

You can invoke **SUB** procedures without using the **CALL** keyword if the **SUB** is declared:

```
DECLARE SUB PrntMsg (Row%,Col%,Msg$)

' Print a message in the middle of the screen.
CLS
PrntMsg 12,40,"Hello!" 'Note the missing parentheses.
END
.
.
.
```

SUB procedures can be used recursively—that is, you can write a **SUB** procedure that calls itself.

DEF FN Procedures

DEF FN procedures are always part of a program's module-level code. For this reason, their use is more limited than that of **SUB** or **FUNCTION** procedures. Like **FUNCTION** procedures, **DEF FN** procedures return single values and are used like built-in BASIC functions:

```
' Function to find log base 10 of a number using
' BASIC's built-in natural logarithm function.
DEF FNLog10 (X)
   FNLog10=LOG(X)/LOG(10.0)
END DEF

INPUT "Enter a number: ",Num
PRINT "10 ^ Log10(";Num;") is" 10.0^FNLog10(Num)
END
```

DEF FN procedure arguments are passed by value (see the following section for more information). The name of a **DEF FN** procedure always begins with **FN**. In addition, **DEF FN** procedures cannot be used recursively and must be defined before they are used. **DEF FN** procedures cannot be called from outside the module in which they are defined.

Passing Arguments to Procedures

BASIC uses two different ways of passing arguments to a procedure. The phrase "passing by reference," used with **SUB** and **FUNCTION** procedures, means the address of each argument is passed to the procedure by placing the address on the stack. The phrase "passing by value," used in **DEF FN** procedures, indicates that the value of the argument is placed on the stack,

rather than the address. Because the procedure does not have access to the variable when an argument is passed by value, the procedure cannot change the variable's value.

Sometimes passing an argument by value to a **SUB** or **FUNCTION** is more convenient. You can simulate a pass by value by using an expression in the **SUB** call or **FUNCTION** invocation:

```
Xcoordinate=Transform((A#))
```

Because `(A#)` is an expression, BASIC calculates its value, `A#`, and passes the address of a temporary location containing the value. An address is still passed, but because it is the address of a temporary location, the action simulates a pass by value.

Recursion

You can use BASIC to write recursive **SUB** or **FUNCTION** procedures (procedures that call themselves). For example, the following program uses a recursive **FUNCTION** to reverse a string of characters:

```
DECLARE FUNCTION Reverse$ (StringVar$)

LINE INPUT "Enter string to reverse: ", X$
PRINT Reverse$(X$)

END
FUNCTION Reverse$ (S$)

   C$ = MID$(S$, 1, 1)
   IF C$ = "" THEN
       ' The first character is null, so return
       ' null--there's no more string left.
       Reverse$ = ""
   ELSE
       ' The reverse of a nonnull string is the first
       ' character appended to the reverse of the remaining
       ' string.
       Reverse$ = Reverse$(MID$(S$, 2)) + C$
   END IF

END FUNCTION
```

Output

```
Enter string to reverse: abcdefgh...tuvwxyz
zyxwvut...hgfedcba
```

`Reverse$` reverses a string by first testing for the simplest case—a null string. If the string is null, then a null string is returned. If the string is not null (there are characters) then `Reverse$` simplifies the problem. The reverse of a non-null string is the rest of the string (`C$`) with the first character of the string concatenated to it. So `Reverse$` calls itself to reverse the rest of the string and when this is done concatenates the first character to the reversed string.

Recursion can use a lot of memory because automatic variables inside the **FUNCTION** or **SUB** must be saved in order to restart the procedure when the recursive call is finished. Because automatic variables are saved on the stack, you may need to increase the stack size with the **STACK** statement to keep from running out of stack space. Use the **STACK** function to determine the maximum stack size you can allocate.

Overlays

If you create very large BASIC programs or programs that dynamically allocate very large arrays, you may occasionally run out of memory during execution. When this occurs, you can reduce memory requirements by using overlays. In an overlaid version of a program, specified parts of the program (known as overlays) are loaded only if and when they are needed. Overlays share the same space in memory at different times during program execution. Only code is overlaid; data is never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to load the code from disk into memory.

You specify overlays by enclosing overlay modules in parentheses in the list of object files that you submit to the LINK utility. Each module in parentheses represents one overlay. For example, the following LINK command line produces three overlays from the seven object files `A` through `I`:

```
LINK A + (B+C) + (E+F) + G + (I)
```

The modules `(B+C)`, `(E+F)`, and `(I)` are overlays. The remaining modules, and any drawn from the run-time libraries, constitute the memory-resident code or root of your program. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can appear only once in a program.

You must compile each module in the program with compatible options. This means, for example, that all modules must be compiled with the same floating-point options.

If expanded memory is present in your computer, overlays are loaded from expanded memory; otherwise, overlays are loaded from disk. You can specify that overlays only be loaded from disk by linking your program with the NOEMS.OBJ stub file.

For more information, refer to the section "Linking with Overlays" in Chapter 18, "Using LINK and LIB."

Index

286 XENIX archives (LIB) 573, 579
80286-specific instructions 544, 559
/? option, ISAMIO utility 389

A

/A option
  BC 558
  HELPMAKE 673
  ISAMIO utility 389
  LINK 588, 590
  NMAKE 632
Active page defined 204
Add-on libraries 567
Advanced feature unavailable error message 441
/Ah option 536–537, 558
Aliases
  ALIAS keyword 430
  variable aliases 64–65
Alphabetic character set 695
Alphabetizing strings 138
Alphanumeric line label 697–698
Alt key combinations, trapping 305–307
Alternate math library for floating-point operations 559, 561–562
Ampersand (&)
  extending operations line (LIB) 582
  long integer type-declaration character 695, 702, 715
AND logical operator 4, 736
AND option, with PUT graphics statement 196–198
And sign (&) *See* Ampersand
Animation
  by rotating color display 176–177
  using GET and PUT graphics statements 192–199
  using screen pages 204–205
Apostrophe (')
  comment character *xxix*, 698
  special character set 696
APPEND mode 92, 98
Arc, drawing 158–160
Argument
  checking number and type with DECLARE statement 50–51
  compared to parameter 44–45
  defined 44
  dummy/placeholder argument defined 89
  list *See* Argument list
  numeric, in LINK 587
  passing by reference
    BASIC 452
    C 452–453
    described 55–56, 743–744
    FORTRAN 453
    Pascal 454
  passing by value
    BASIC 451
    C 452
    described 56–57, 743–744
    FORTRAN 453
    Pascal 454
  passing to procedure
    constants 45–46
    described 44–45, 743–744
    expressions 46
    variables 47–50
Argument list
  defined 45
  variable type-declaration methods 47
Arithmetic
  algebraic expression, counterpart in BASIC 733
  floating-point math *See* Floating-point math operations
  modulo arithmetic 733–734
  operations
    order of execution 731–732
    precedence over relational operation 734
  run-time errors 734
Arithmetic operators 733–734
Array
  defined 717
  described 717–718
  dynamic *See* Dynamic array
  in mixed-language programming 463–470

Array *(continued)*
numeric 535–536, 720
passing to procedure
elements of array 48
entire array 47–48
finding size of array 49
static *See* Static array
storage
comparing BC and QBX 464
described 725–726
in column-major order, default 468
in expanded memory *See* Ch. 3 in the *Getting Started* booklet
in row-major order, /R option 560
string *See* String array
variables
declaring 717–718
defined 712
Array-bound functions 49
Arrow keys
described *xxx*
trapping 304
AS clause, variable type declaration 66, 715–716
AS STRING statement 134, 715
ASC function 134
ASCII code
comparing strings 137–138, 739
determining corresponding character 134
ASCII file
See also Sequential file
formats for Help files 671–672
ISAM import/export utility 389–391
Aspect ratio
defined 162
drawing shapes to proportion 162–164
ellipse drawing 157
Assembly language
See also Mixed-language programming
code, listing with /A option 558
language equivalents for routine calls 420
MASM calling BASIC 498–500
procedures *See* Assembly language procedure
using with custom run-time modules 667–668
Assembly language procedure
called from BASIC 484–486
entering 477
exiting 483
local data space 477
overview 475–476
parameters, accessing 479–481
register values 478–479
Assembly language procedure *(continued)*
returning a value 481–482
segment model 486–488
setting up 476–477
Assignment symbol (=), special character set 696
Asterisk (*)
copy command symbol (LIB) 580
multiplication symbol 696
wildcard character (NMAKE) 637
At sign (@)
currency type-declaration character 696, 702, 715
in filename to be used with LINK or LIB 563
suppressing command display (NMAKE) 638
Automatic variable 65, 726–727
Axis in Presentation Graphics *See* Presentation Graphics toolbox

B

B_OnExit routine
mixed-language programming 474–475
Quick library 627–628
/B option (ISAMCVT utility) 391
B_OVREMAP routine 613
/BA option (LINK) 588, 591
Background color *See* Color
Backslash (\), integer division symbol 696, 733
Backup files *See* NMAKE
Bad file name error message 94
BALLPSET.BAS 199–202
BALLXOR.BAS 202–203
Bar chart *See* Presentation Graphics toolbox
Bar-graph generator, sample program 205–210
BAR.BAS 205–210
.BAS filename extension
See also BASIC source file
BC 558
BASIC
array handling 463–470
calling assembly language procedure 484–486
calling BASIC from another language
See also specific language
overview 443–444
calling C
far strings 501–503
near strings 432–435
calling conventions 450
calling FORTRAN
far strings 505–507
near strings 435–437

BASIC *(continued)*
calling Pascal
far strings 510–512
near strings 437–439
character set 695–696
file I/O in mixed-language programming 442
interface to other languages 429–432
language equivalents for routine calls 420
naming conventions 420–423, 450
numeric data types, equivalents 454–455
passing arguments 451–452
passing parameters 425–427
passing strings
from BASIC 459–460
to C 460–461
to FORTRAN 462
to Pascal 463
string format 456–457
user-defined data type 470–471
BASIC Compiler *See* BC
BASIC program *See* Program
BASIC program line *See* Program line
BASIC source file
creating Quick library 616
filename extension 557–558
naming 93–94
BASIC statement *See* Statement
BASIC toolbox files 568
Batch mode (LINK) 591
Baud defined 115
BC (BASIC Compiler)
array storage, compared to QBX 464
command *See* BC command
DGROUP storage as default 404
filenames 557–558
overview 555
warnings, suppression of 560
wildcard character not accepted 637
BC command
compiling with 556–557
options
described 558–562
new to version 7.0 555
BCD data types 547
BEEP statement 527
BEGINTRANS statement
ISAM 382, 383
protected mode 520
BestMode function 231, 237, 243
Binary access compared to random access 110–111
Binary-coded-decimal (BCD) data types 547
Binary file
access, compared to ISAM 322
I/O 102, 110–111
Binary format *See* Microsoft Binary format
Binary number, converting to hexadecimal 183
Bit planes in different screen modes 188–189, 193
Blank character
See also Space
special character set 695
BLOAD statement
far-string processing 404
in protected mode 520, 526
Block IF...THEN...ELSE statement 8–10, 30, 31
BOF function (ISAM) 343–344
BOF keyword 520
BOOKS.BAS 331
Boolean expressions 4–6, 735, 736
Bouncing ball, sample programs 199–203
Box
drawing 153–154
filling 154
painting with color or tiles 177–180
Brackets ([]), special character set 696
Branching 6, 315
BSAVE statement
far-string processing 404
in protected mode 520, 526
BSEDOSFL.BI, OS/2 include file 523
BSEDOSPC.BI, OS/2 include file 523
BSEDOSPE.BI, OS/2 include file 523
BSESUBMO.BI, OS/2 include file 523
Btrieve
code, converting to ISAM code 395–399
database, converting to ISAM file format 391–393
Buffer
communications port 116, 303, 559
music buffer, trapping 560
setting for ISAM 377–378
BUILDRTM utility 527, 663–666
BYVAL keyword
declaring parameter 427, 430–431
passing arguments 451

C

C (language)
array
declaring 467
indexing 466

C (language) *(continued)*
calling BASIC 444–446, 503–505
calling conventions 450
functions incompatible with BASIC 441
language equivalents for routine calls 420
memory allocation in mixed-language programming 440
naming conventions 421–423, 450
numeric data types, equivalents 454–456
passing arguments 452–453
passing parameters 427
passing strings to BASIC 461–462
string format 457–458
/C option
BC 559
HELPMAKE 673
ISAMIO utility 390
NMAKE 632
CAL.BAS 116–121
Calendar, perpetual, sample program 116–121
CALL ABSOLUTE statement 521, 526
CALL INT86 statement 521
CALL INT86OLD statement 521
CALL INT86X statement 521
CALL INTERRUPT statement 521
CALL statement
calling a SUB procedure 43–44
in protected mode 520
Calling
See also specific language; Mixed-language programming
custom run-time module 666–667
FUNCTION procedure 42–43
speed, improving procedure calls 548–549
SUB procedure 43–44
Calling conventions 423–425, 450–451
CALLS statement
mixed-language programming 432
passing BASIC argument 451–452
Caps Lock key, trapping 305–306
Caret (^)
escape character (NMAKE) 637
special character set 696
Carriage-return key *See* Enter character
Carriage-return line-feed sequence (CR-LF)
field delimiter 95–96
filter, sample program 31–34
CASE clause
See also SELECT CASE statement
syntax 12–14
CASE ELSE clause
See also SELECT CASE statement
syntax 14–15
Case sensitivity/insensitivity
BC 557
converting lowercase/uppercase in string 143
DOS filename 94
HELPMAKE options 673
LIB 576
LINK 596
NMAKE options 664
CDECL keyword 419, 429
CGA *See* Color Graphics Adapter
CHAIN statement 69–71, 668
Chaining of programs 69–71
Character
case *See* Case sensitivity/insensitivity
comment characters *See* Comment
data type-declaration 695–696, 702, 715
determining corresponding ASCII code 134
escape character (NMAKE) 637
reading specified number of characters from file 101–102
repeating 142–143
retrieving from any part of string 140–142
sending to/printing from modem 130–132
special characters *See* Special character
string defined 133
width of, changing 81
wildcard character (NMAKE) 637
Character set used by BASIC 695–696
Charts *See* Presentation Graphics toolbox
CHECK.BAS 29–31
Checkbook balancing, sample program 29–31
Checkpoint, implicit (ISAM) 559
CHECKPOINT keyword 520
CHR$ function 134, 217
CHRTASM.OBJ 223
CHRTB.BAS 223, 266, 568, 625
CHRTBEFR.QLB 223
Circle, drawing 156–157
CIRCLE statement
aspect ratio, automatic adjustment 157
color argument 172
drawing shapes
arc 158–160
circle 156–157
ellipse 157–158
pie shape/wedge 161
STEP keyword 152
syntax 156

CLEAR statement 440
CLOSE statement 94, 336, 520
Closing a data file 94–95
CLS 2 statement 82
/CO option (LINK) 588, 591
Code
  compiling *See* Compiling
  debugging *See* Debugging
  module-level code *See* Module-level code
  object code, listing with /A option 558
  sample code, copying to file/executing 518
  segments *See* LINK
CodeView debugger
  LINK 591
  OS/2 programs 529
Coercion, data type (ISAM) 334–335
Colon (:)
  double (::) as separator character (NMAKE) 639
  special character set 696
Color
  argument, in graphics statements 172
  background/foreground, changing 172–174
  CGA screen modes 171
  changing with PALETTE statements 175–176
  color/clarity tradeoff 171
  EGA 175
  in Presentations Graphics *See* Presentation Graphics toolbox
  multicolor patterns 185–190
  painting shapes 177–179
  pixels 150
  sample programs
    illusion of movement 176–177
    showing color combinations 174–175
  VGA 175
Color Graphics Adapter (CGA)
  color palettes 173
  Presentation Graphics toolbox 223
  screen modes supported 171
COLOR statement
  graphics screen mode 1 173
  in protected mode 521
COLORS.BAS 174–175
Column
  advancing to specific column 80–81
  changing number of columns 81
  screen columns described 78
Column chart *See* Presentation Graphics toolbox
COM ports 113
COM statement, event trapping 303, 560
Comma (,)
  field delimiter 96
  in INPUT statement 84
  in LINK syntax 584
  in PRINT statement 79
  special character set 696
Comment
  characters
    export list for custom run-time module 662
    NMAKE 636
    nonexecutable program statement 698
  defined 698
  notation convention for manual *xxix*
COMMITTRANS statement
  ISAM 382, 383
  protected mode 520
COMMON block
  in chaining 70
  in mixed-language programming 472
COMMON SHARED statement 62
COMMON statement
  declaring global variable 722
  defining fixed-length string 135–136
  nonexecutable statement 698
  not allowed in procedure definition 41
  passing variables in chain 70
  SHARED attribute 60
  sharing variables across modules 62–64
Communications
  event trapping 560
  through serial port 115–116
Communications buffer 303, 559
Compile-time errors, listing 560
Compiler *See* BC (BASIC Compiler)
Compiling
  BC command 556–557
  from command line
    error-handling options 299
    event-trapping options 316–317
    string handling 720
  from QBX
    BC options available 561
    string handling 720
  modules 272
  OS/2 programs 527–528
  speed, optimizing programs 544–545
Concatenation
  defined 136
  of strings 136–137, 739
CONS: device 113

CONST statement
declaring symbolic constant 134, 722
nonexecutable statement 698
Constant
defined 707
literal constant, defined 707
local constant 723
numeric constant 708–712, 728
passing to procedure 45–46
program speed considerations 546
string constant 133–134, 708
symbolic *See* Symbolic constant
Control-flow structure
See also Decision structure; Loop
optimizing for speed 547–550
overview 3
Conventions
filenames 93–94
programming conventions used in manual *xxxi–xxxii*
typographic conventions used in manual *xxix–xxxi*
Coordinates
See also Pixel
comparing graphics and text-mode coordinates 150
defining with WINDOW statement 166–169
order of coordinate pairs 151, 170
physical, defined 166
relative, designating 152–153
screen coordinates, overview 148–150
translating physical and window coordinates with PMAP function 170–171
window coordinates defined 167
/CP option (LINK) 590
CREATEINDEX statement
ISAM 337–338
protected mode 520
CR-LF *See* Carriage-return line-feed sequence
CRLF.BAS 31–34
CSRLIN function 89
Ctrl+Alt+Del, trapping 307
Ctrl+Break, enabling 559
Ctrl+C
terminating library session 574
terminating LINK operation 583
Ctrl key combinations, trapping 305–307
Cube, rotating, sample program 204–205
CUBE.BAS 204–205
Currency constant 709, 712
Currency data type
at symbol (@) as suffix 696, 702, 715
described 547, 706
ISAM 334
program speed considerations 547
Cursor
graphics cursor 152, 170
text cursor 87–89
Cursor movement keys *See* Arrow keys
Custom run-time module
assembly language programming 667
BUILDRTM utility
creating default run-time module/library 665
files created by 666
invoking 663–664
messages 665
creating
by BUILDRTM utility 666
steps 661
data object handling 667
DGROUP references 668
executable file, creating 666–667
export list, creating 662–663
EXPORTS directive 663
import object file 666
LIBRARIES directive 663
linking 666–667
OBJECTS directive 663
overview 661
programming considerations 667–668
run-time library 666
source files, compiling 662
CVD function, converting to Microsoft Binary format 560
CVS function, converting to Microsoft Binary format 560

D

/D option
BC 559
HELPMAKE 676
ISAM 377, 391, 550
NMAKE 632
Dash (-)
See also Minus sign (-)
option indicator
BC command 558
HELPMAKE 673
turning off error checking (NMAKE) 638

Data
  adding
    to random-access file 103–105
    to sequential file 96–100
  external data defined 471
  file *See* Data file
  handling in mixed-language programming 450–454
  in Presentation Graphics *See* Presentation Graphics toolbox
  managing for speed 546–547
  numeric 454–456
  passing by address 471
  reading from sequential file 97–98, 100–102
  string data *See* String
  transfer with ISAM 342
  trapping in communications buffer 303
  types *See* Data types
  window *See* Presentation Graphics toolbox
Data file
  appending records 92
  closing 94–95
  creating 91
  defined 90
  filenames 93–94
  numbers 92–93, 94
  opening 91–92
  overwriting 91
  random-access file *See* Random-access file
  sequential file *See* Sequential file
Data segment *See* LINK
DATA statement, nonexecutable 698
Data types
  currency numbers
    compared to binary-coded-decimal 547
    described 702, 706
  declaration characters 695–696, 702, 715
  fixed-length string 701
  floating-point numbers 702, 703–705
  integer numbers 702–703
  ISAM 333–334
  OS/2, simulating types in include files 523–526
  suffixes *See herein* declaration characters
  user-defined
    described 706
    passing in mixed-language programming 470–471
  variable-length string 701
Data window *See* Presentation Graphics toolbox
Date/time functions, add-on libraries 567
db/LIB file, converting to ISAM format 391–393
Debug code, generating with /D option 559
Debugging
  including line numbers for SYMDEB 594
  preparing for using CodeView 529, 591
Decimal numeric constant 708, 709
Decimal point (.), special character set 696
Decision structure
  defined 6
  IF...THEN...ELSE statement
    block 8–10
    single line 6–8
  SELECT CASE statement 10–16
DECLARE statement
  checking arguments 50–51
  in include file 53–55
  nonexecutable statement 698
  not allowed in procedure definition 41
  QBX
    programs developed outside of 52–53
    when not generated 51
  syntax in mixed-language programming 429–430
  use with multiple modules 39, 271
.DEF filename extension (LINK) 585
DEF FN function 38–39, 41
DEF FN procedure
  compared to FUNCTION procedure 725, 742
  described 743
  optimizing execution speed 562
  variables 724–725
DEF SEG statement, in protected mode 521, 526–527
DEFSTR statement 134, 716
DEF*type* statement
  declaring variables 716–717
  nonexecutable statement 698
DELETE statement (ISAM) 342
DELETE*element* keyword 520
DELETEINDEX statement (ISAM) 374–375
DELETETABLE statement (ISAM) 374–375
Description file *See* NMAKE
Device driver *See* Ch. 3 in the *Getting Started* booklet
Device I/O
  compared to file I/O 114
  devices supported by BASIC 113–114
DGROUP
  defined 531, 719
  overview 403
  references in mixed-language programs 668

DGROUP *(continued)*
types of data stored 534–536
unused space, return 407
DIM statement
declaring fixed-length string 135–136
nonexecutable statement 698
SHARED attribute
declaring variables 60–62, 722–724
not allowed in procedure definition 41
Direction keys *See* Arrow keys
Directives
custom run-time module 663
NMAKE 650–652
Directory search, sample program 72–75
Division symbols 696, 733
.DLL filename extension 530
DO...LOOP statements
compared to WHILE...WEND statement 23
exiting a loop 29
loop test 27–28
syntax 24–27
/DO option (LINK) 588, 591
Dollar sign ($)
NMAKE macros 642–643
string type-declaration character 134, 695, 715
DOS filename conventions 93–94
DOSBEEP function 527
DOUBLE data type (ISAM) 334
Double-precision numeric constant 710, 711
Double-precision numeric data type 103, 695, 702
DRAW statement
described 190–192
use of floating-point math package 541
Drawing
See also Animation; Graphics
arc 158–160
aspect ratio, adjusting 162–164
box 153–154
circle 156–157
ellipse 157–158
lines 151–153, 155–156
macro language 190–192
pie shape/wedge 161
plotting points 150–151
proportionate drawings 162–164
/DS option (LINK) 590
DTFMTAP.LIB 567
DTFMTAR.LIB 567
DTFMTEP.LIB 567
DTFMTER.LIB 567, 623
Dumb terminal, sample program 130–132
Duplicate definition error message 626
Dynamic array
compared to static array 547
described 725–726
huge dynamic array 536–537, 721
numeric 534, 536, 719
string 535, 720
using available memory with /Ah option 558
Dynamic-link library 526, 530
$DYNAMIC metacommand 726

E

/E option
BC 299, 559
HELPMAKE 674
ISAMIO utility 389
LINK 588, 592
NMAKE 633
EDPAT.BAS 217–222
EGA *See* Enhanced Graphics Adapter
Ellipse, drawing 157–158
ELSE clause 7
ELSEIF clause 9
EMS *See* Expanded Memory Specification
Emulator library 561
END FUNCTION statement 41
END SELECT clause 12–15
END SUB statement 41
Enhanced Graphics Adapter (EGA)
color changes with PALETTE statement 173, 175
Presentation Graphics toolbox 223
ENTAB.BAS 721–722
Enter character, special character set 695
EOF function
device compatibility 114, 116
ISAM 343–344
not supported in protected mode 520
Equal sign (=)
assignment symbol 696
relational operator 4, 137, 696, 734
EQV logical operator 4, 736
ERL function 698
ERR function 277–278, 293–294
Error during run-time initialization error message 561
ERROR ERR statement 279, 281, 292
Error handling
See also Error trapping
compile-time errors, listing 560

Error handling *(continued)*
defined 275
delay in handling 293–294
/E option 299
in mixed-language programming 442–443
invocation path defined 281
loop, error occurring in 280
module-level
examples 281–284
multiple modules 285–286
trap 277
NMAKE, turning off error checking 638
OS/2 include file 523
overview of procedure 275–276
Presentation Graphics *See* Presentation Graphics toolbox
procedure-level 277, 281–284
routine *See* Error-handling routine
turning off 295
unanticipated error 286–295
using for program control 296–298
/X option 299
Error-handling routine
errors occurring within routine 292
exiting 278–281
guidelines for complex programs 292
multiple-modules, example 285–286
procedure/module level, examples 281–284
searching the invocation path for 286–295
writing 277–278
Error message
See also specific message
HELPMAKE 689–692
ISAM 399–401
OS/2 include file 523
Error trapping
See also Error handling
ERL function 698
minimizing generated code 542–543
setting the error trap 277
user-defined errors 298
Escape character (NMAKE) 637
Event handling
See also Event trapping
defined 298
in mixed-language programming 442
polling compared to trapping 298
routine *See* Event-handling routine
trapping *See* Event trapping
Event-handling routine
error occurring within routine 292
event occurring within routine 315
location 303
EVENT OFF statement 312–313
EVENT ON statement 312, 313, 316
event STOP statement 314
Event trapping
across modules 315–316
compared to polling 298–299
keystroke trapping 303–307
minimizing generated code 542–543
music event 308
overview of procedure 298–299
program speed considerations 551
Quick library, use with module which traps events 620
statements used for trapping 303
suspending specific event traps 313–315
trappable events, listing of 303
turning off selectively 313
user-defined event 310–312, 560
/V option 316, 560
/W option 316, 560
Exclamation point (!)
NMAKE 639, 650
single-precision type-declaration character 695, 702, 715
.EXE file
filename extension (LINK) 585
relational operations, differing results using QBX 734–735
Executable file
exefile field (LINK) 601
for use with ISAM 379–380
minimizing size 544
packing (LINK) 592
Executable statement defined 698
EXIT DO statement 29
EXIT FUNCTION statement 41
EXIT SUB statement 41
Expanded memory *See* Ch. 3 in the *Getting Started* booklet
Expanded Memory Specification (EMS)
ISAM considerations 376, 378–379, 381
LINK 613
unused space, return 407
Exponentiation symbol (^), special character set 696

Expression
  Boolean 4–6, 735
  data type conversion 728–729
  defined 731
  passing to procedure 46
  printing list of 79
  string expression *See* String expression
Extended key, trapping 306
Extended memory *See* Ch. 3 in the *Getting Started* booklet
Extended run-time module (OS/2) 527
Extension *See specific extension*; Filename extension

F

F1 through F12 keys *See* Function keys
/F option
  ISAMIO utility 390
  LINK 588, 592–593
  NMAKE 633
Factorial function 67–68
False expression 4–6, 734, 736
Far call (LINK) 592–593, 595
Far heap 531, 534, 719
far keyword 427–428, 444–445
Far memory 531, 719
Far string
  compared to near string 403–404
  data structure described 414–415
  defined 403
  direct processing 404–406
  enabling with /Fs option 404, 559, 561
  memory space
    allocation 414–415, 534
    calculating 406–407
    maximizing 410–413
  mixed-language programming 417, 489
  passing to procedures in other languages 408–410
  pointers 408–410
  program speed considerations 547
  using with previous version of BASIC 413
fastcall keyword 425
Feature stubbed out error message 539
Field
  defined 90
  in random-access file 103
  in sequential file 95–96
FIELD statement 103–104
File
  access modes 110–111
  binary access 110–111
  data file *See* Data file
  description file *See* NMAKE
  executable *See* Executable file
  I/O, compared to device I/O 114–115
  ISAM *See* ISAM
  library *See* Library file
  naming *See* Filename
  numbering 92–93
  object *See* Object file
  pointer *See* File pointer
  random-access *See* Random-access file
  reading specified number of characters from 101–102
  response file *See* LIB
  sequential file *See* Sequential file
  stub file *See* Stub file
  updating *See* NMAKE
File pointer
  moving with GET and PUT 110
  positioning with SEEK 111
FILEATTR function
  ISAM 336–337
  protected mode 520
Filename
  BASIC naming conventions 93–94
  LINK, LIB, use of at sign (@) 563
  run-time module conventions 529
  scanning directory for, sample program 72–75
  specifications for use with BC command 557–558
Filename extension
  BC 558
  defined 557
  LINK 585
Fill pattern *See* Presentation Graphics toolbox
Filling *See* Painting
FINANCAP.LIB 567
FINANCAR.LIB 567
FINANCEP.LIB 567
FINANCER.LIB 567, 623
Financial functions, add-on libraries 567
Fixed-length string
  comparing with variable-length string
    using RTRIM$ function 140–141
    with relational operators 137–138

Fixed-length string *(continued)*
declaring 715–716
described 135–136
passing in mixed-language programming 491, 517–518
Fixed-point numeric constant 709
Floating-point math operations
accidental use of 542
alternate math library 559, 561–562
division symbol (/) 733
in-line instructions 559, 561
Floating-point number 703–706
Floating-point numeric constant 710
Floating-point value, conversion 729
FLPT.BAS 703–705
Font
graphics 224, 266
text 252
Font toolbox file 568
FONTB.BAS 266, 568
FOR...NEXT loop
described 16–20
exiting 20–21
suspending program execution 21
FOR...NEXT statement 17
Foreground color *See* Color
FORTRAN
array
declaring 467
indexing 466
calling BASIC 443–444, 447–449, 508–510
calling conventions 450–451
language equivalents for routine calls 420
naming conventions 420–423, 450
numeric data types, equivalents 454–455
passing arguments 453
passing parameters 425–427
passing strings to BASIC 462
record data type 470
string format 458
fortran keyword 421, 431–432, 443–445
Forward reference defined 51
Forward slash *See* Slash (/)
/FPa option (BC) 559
/FPi option
BC 559
compiling routine for Quick library 617
/FP*method* option (BUILDRTM) 664
Fractal, sample program 211–217
Frame number (LINK) 607
FRE function
adjusting size of stack for recursive procedure 68–69
determining available memory 534
near/far string storage space 406–407
FREEFILE function 92–93
/Fs option
BC 404, 559
BUILDRTM 664
compiling routine for Quick library 617
module compatibility 561
Function
See also specific function
intrinsic 738
use in protected mode 520–522
user-defined 738
FUNCTION...END FUNCTION statement, not allowed in procedure definition 41
Function keys, trapping 304, 560
FUNCTION procedure
calling 42–43
compared to DEF FN procedure 38, 742
DECLARE statement
generating in QBX 51
syntax in mixed-language programming 429–430
definition 40–41
described 742
improving speed of calls 548–549, 562
in multiple-modules 39
listing in custom run-time module 663
recursive 39, 744
syntax 40–41
variables
changing 38–39
local by default 38
Functional operators 738

G

/G2 option, generating code for 80286 processor 544, 559
/Gc option, compiling C module 431–432
GENERAL.BAS 568
GET statement
binary-access file I/O 110–111
graphics
animation 199
saving images 193–195
random-access file I/O 109
STEP keyword 152

GETINDEX function (ISAM) 341, 520
GETINDEX$ keyword 520
GetSpace$ procedure 441
Global symbol, listing (LINK) 595
Global symbolic constant 722, 725
Global variable
  declaring with SHARED attribute 60
  DEF FN function 38
  GOSUB subroutine 37
  scope 722, 725
GOSUB subroutine 36–38, 420
Granularity defined 538
Graphics
  See also Presentation Graphics toolbox
  animation *See* Animation
  color *See* Color
  computer requirements to run graphics examples 147–148
  coordinates
    See also Coordinates
    comparing graphics-mode and text-mode 150
  drawing *See* Drawing
  in protected-mode program 527
  macro language 190–192
  overview 147
  plotting points 150–151
  screen mode 148
Graphics cursor 152, 170
Graphics fonts 224, 266
Graphics statements
  See also specific statement
  color argument 172
Graphics viewport
  defining 164–166
  physical/window coordinates 166–167
Greater than or equal to (>=), relational operator 4, 137, 734
Greater than symbol (>)
  redirecting output 78
  relational operator 4, 137, 734
  special character set 696
Grid 149

H

/H option
  BUILDRTM 664
  HELPMAKE 674, 676
  ISAMIO utility 389
/HE option (LINK) 588, 593
Heap defined 719
Help
  See also HELPMAKE; Ch. 4 in the *Getting Started* booklet
  online Help for mixed-language programming 518
  overview 669
Help database
  See also Help text file; HELPMAKE
  contents 669
  context
    conventions 677–679
    defined 669
  control character, application-specific 681–682
  converting to Help text file 672
  creating 677
  decoding 672
  formats
    ASCII, minimally formatted 682, 686
    QuickHelp 682–685
    RTF 682, 687–688
  naming with /O option 674
  subjects *See herein* context
  topic text defined 669
/HELP option
  LIB 576
  NMAKE 633
Help text file
  See also Help database; HELPMAKE
  context
    conventions 677–679
    local contexts 681
  converting to Help database 672
  cross-references to Help database 670
  encoding 672
  explicit cross-reference
    described 670
    format 680
format
  ASCII formats 672, 679–680
  Quick Help 671, 679–680
  RTF 671, 679–680
formatting flags 670–671
hyperlink
  See also herein explicit cross-reference
  defined 670
implicit cross-reference 670
:l command 681
:n command 681
:p command 681

HELPMAKE
creating a Help database 677
error messages 689–692
format of Help text 679–680
invoking 672–673
local contexts 681
options
decoding 675–676
encoding 673–675
overview 669
Hercules monochrome graphics 223
Hexadecimal
argument in LINK 587
converting binary to hexadecimal number 183
numeric constant 708, 709
/HI option (LINK) 590
HIMEM.SYS device driver *See* Ch. 3 in the *Getting Started* booklet
Huge dynamic array 536–537, 719, 721
Hyphen *See* Dash; Minus sign

I

/I option
ISAMIO utility 389
LIB 576
NMAKE 633
/Ib option (ISAM) 376
/Ie option (ISAM) 376
IEEE format floating-point numbers 703
IF...THEN...ELSE statement
block form 7, 8–10
in BASICA 6–7
IF...THEN statement
described 6–7
line label restrictions 697
.IGNORE: pseudotarget (NMAKE) 653
/Ii option (ISAM) 377
Illegal function call error message 196
IMP logical operator 4, 736
IMPORT.OBJ file 666
In-line instructions for floating-point math operations 559, 561
Include file
OS/2 523
passing variables in a chain 70–71
uses described 53–55
$INCLUDE metacommand 53, 55, 70, 523
Index *See* ISAM
INDEX.BAS 121–130
Indexed Sequential Access Method *See* ISAM
Indexing a random-access file, sample program 121–130
/INF option (LINK) 589, 594
Inference rules *See* NMAKE
INKEY$ function 87
INP function, unavailable in protected mode 521
Input
See also I/O
binary input 102
entering input from keyboard (standard input) 84–90
prompts 85
INPUT$ function
binary input 102
compared to INKEY$ function 87
described 86–87
reading data
binary-access file 111
communications device 116
sequential file 101–102
Input past end of file error message 99
INPUT statement
compared to INKEY$ function 87
compared to INPUT$ function 86
compared to LINE INPUT statement 85–86
described 84–85
use of floating-point math package 542
INPUT # statement 97–98
Input/output *See* I/O
INSERT statement
ISAM 342
protected mode 520
INSTR function 138–139
Insufficient heap space error message 440
Insufficient memory error message (LIB) 576
Integer
data type
described 702–703
ISAM 333
suffix 695
division 696, 733
long integer *See* Long integer
numeric constant 708
program speed considerations 546
Intel 80286, /G2 compiler option 544
Intel-format library (LIB) 573, 579
INTERFACE statement 443
Intrinsic function defined 738

I/O
See also Input; Output
binary file I/O 110
comparing file and device I/O 113–114
optimizing for speed 550
overview 77
IOCTL$ function, unavailable in protected mode 521
IOCTL statement, unavailable in protected mode 521
ISAM
386MAX.SYS device driver *See* Ch. 3 in the *Getting Started* booklet
BASIC keywords not supported for use with ISAM 520
BEGINTRANS statement 382, 383
block processing statements 382
BOF function 343–344
Btrieve
code, converting to ISAM code 395–399
database, converting to ISAM file format 391–393
buffers 377–378
checkpoint, implicit 559
CLOSE statement 336
coercion, data types 334–335
column
defined 323–324
naming 333
combined index
creating 340
defined 326
COMMITTRANS statement 382, 383
compacting a database 394
compared to other types of file access 321–322
CREATEINDEX statement 337–338
current index
changing 343
defined 327
determining 341
current position
defined 327
description 343
current record
defined 327
setting 343–344, 354–355
/D option 377, 550, 559
data
deleting 342
transferring 342
ISAM *(continued)*
data dictionary defined 327
data types
coercion 334–335
specifying 333–334
database
compacting 394
defined 324
integrity 386–387
repairing 393–394
restoring 382
defined 319
DELETE statement 342
DELETEINDEX statement 374–375
DELETETABLE statement 374–375
deleting data 342
EMS *See herein* Expanded Memory Specification
engine
defined 327
described 328
EOF function 343–344
error messages 399–401
executable files used with ISAM programs 379–380
Expanded Memory Specification (EMS)
See also Ch. 3 in the *Getting Started* booklet
option 376
overview 378–379
practical considerations 381
field defined 323
file
allocation and growth 329
attribute, determining 336–337
conversion utility 391–393
defined 327
parts, described 328
FILEATTR function 336–337
focus defined 327
GETINDEX function 341
/Ib option 376
/Ie option 376
See also Ch. 3 in the *Getting Started* booklet
described 376
/Ii option 377
in QBX 375–377
index
combined *See herein* combined index
creating 337–338

ISAM *(continued)*
index *(continued)*
current *See herein* current index
defined 324–326
deleting 374–375
described 329–331
NULL *See herein* NULL index
restrictions 341
specifying 337–338
unique *See herein* unique index
indexed value defined 326
INSERT statement 342
insertion order defined 326
ISAMIO 389–391
/Ix:*number* option 559
key value defined 326
memory
expanded memory *See herein* Expanded Memory Specification
management 559
reserved for buffers 376
MOVE*dest* statement 343–344
MOVEFIRST statement 344
MOVELAST statement 344
MOVENEXT statement 344
MOVEPREVIOUS statement 344
multiple files 388
multi-table database, sample 366–374
naming conventions 375, 714
non-NULL indexes 377
NULL index
default 337
defined 326
OPEN statement 335–336
optimizing I/O for speed 550
overview 319–320
presentation order defined 326
PROISAM.EXE 375–377
PROISAMD.EXE 375–377
protected mode, not supported in 520
RAMDRIVE.SYS device driver *See* Ch. 3 in the *Getting Started* booklet
record
access, table/index model 329–331
current *See herein* current record
data manipulation statements 342–343
defined 323
order, defined 326
subordering 339–340

ISAM *(continued)*
record variable
defining 337
subset of table column 387–388
repairing a database 393–394
restoring a database 382
RETRIEVE statement 342
ROLLBACK statement 382, 383
row defined 323
run-time error messages 399–401
sample database 332
sample program 345–354
save points 383–384
SAVEPOINT statement 382, 383–384
SEEKEQ statement 354
SEEKGE statement 354
SEEKGT statement 354
seeking a string 355
SEEK*operand* statement 343
SETINDEX statement 337–338, 343
statements used 320–321
string comparison 355–356
subordering of records 339–340
supported by MS-DOS 319
table
closing 336
column *See herein* column
defined 323
deleting 374–375
described 329–331
opening 335–336
task
comparison of ISAM and random-file approaches 322–323
outlined 320–321
terms defined 323–327
TEXTCOMP function 355–356
transaction
block, specifying 382
log 383
statements 382
transferring data 342
TSR *See* ISAM TSR
TYPE...END TYPE statement 320–321
unique index
creating 339
defined 326–327
UPDATE statement 342
using with compiled programs 379–380

ISAM *(continued)*
utilities
ASCII import/export (ISAMIO) 389–391
compacting a database (ISAMPACK) 393
file conversion (ISAMCVT) 391–393
repairing a database (ISAMREPR) 393–394
ISAM TSR
program, installation and deinstallation order 381–382
removing from memory 377
ISAMCVT utility 391–393
ISAMIO utility 389–391
ISAMPACK utility 394
ISAMREPR utility 393–394
/Ix:*number* option 559

J

Joystick trigger
event trapping 303, 560
unavailable in protected mode 527

K

Key
See also Arrow keys; Function keys
trapping
preassigned keystrokes 304
user-defined keystrokes 305–307
KEY statement
described 305
event trapping 303, 560
Keyboard
input functions and statements 86–87
input only mode 114
KYBD: device 114

L

/L option
BUILDRTM 664
HELPMAKE 674
loading Quick library 615, 616, 618, 625
Label *See* Line label
Languages *See specific language*; Mixed-language programming
LBOUND function 49
LCASE$ function 143
LEFT$ function 140
Legend *See* Presentation Graphics toolbox
LEN function 104–105
Less than or equal to (<=) 4, 137, 734
Less than symbol (<)
relational operator 4, 137, 734
special character set 696
Letter
See also Character
case sensitivity *See* Case sensitivity/insensitivity
converting uppercase/lowercase 143
/LI option (LINK) 589, 594–595
LIB
286 XENIX archives, converting to DOS format 579
case sensitivity when comparing symbols 576
combining libraries 579
command
described 573, 577
fields 574
processing described 578
symbols listed 577
consistency check 575
cross-reference listing file 581
extended dictionary, not generating 576
Intel-format library, converting to DOS format 579
library file, creating 575
list file prompt 582
naming libraries 581
object module
adding 578–579
copying into object file 580
defined 578
deleting 579
moving to object file 580
replacing 579–580
object-module library
creating 620
defined 573
operations
line, extending 582
prompt 582
options, list 576
output library prompt 582
overview 573
page size, specifying 577
prompts 581–582
response file
defined 573
described 582–583

LIB *(continued)*
syntax 574
terminating a library session 574
.LIB file
creating Quick library from 623
described 564–567
.LIB filename extension 564, 585
Library
See also LIB
add-on libraries 567
alternate math library for floating-point operations 559, 561–562
combining two libraries 579
date/time functions, supplied by Microsoft BASIC 567
default
creating with BUILDRTM 665
ignoring in LINK search 595
substituting stand-alone for run-time library 560
defined 563
dynamic-link library 526, 530
emulator library 561
file *See* Library file
financial functions, supplied by Microsoft BASIC 567
linking in mixed-language programming 428
naming 581
object module
adding 578–579
alignment 577
copying into object file 580
deleting 579
moving to object file 580
replacing 579–580
object-module libraries 564–567, 620
overview 563
page size 575, 577
public symbols, listing 581
Quick library *See* Quick library
run-time library *See* Run-time library
stand-alone library
described 565–566
substituting for default run-time library 560
Library file
creating (LIB) 574–575
linking (LINK) 601
listing for custom run-time module 663
Lightpen
See also PEN statement
event trapping 303, 560
unavailable in protected mode 527
Line
BASIC program line syntax 696–699
drawing lines *See* Drawing
end of line character 31, 695
extending operations line (LIB) 582
QBX program line length 699
styles in Presentation Graphics *See* Presentation Graphics toolbox
wrapping on screen 79
Line chart *See* Presentation Graphics toolbox
Line continuation symbol (_) 696, 699
Line identifier 696
LINE INPUT statement
compared to INKEY$ function 87
compared to INPUT$ function 86
compared to INPUT statement 85–86
described 85–86
LINE INPUT # statement 100–101
Line label
alphanumeric 697–698
removing for optimal program speed 551
Line number
described 697
including in map file (LINK) 594–595
zero (0), in error-handling routine 277
LINE statement
color argument 172
drawing
box 153–154
lines 155–156
STEP keyword 152
syntax 151
LINK
batch mode 591
case sensitivity 596
code segment
disabling packing 596
packing 596–597
command-line syntax 584
custom run-time module 666
data segment
aligning 590
packing 597
debugging, preparing for use with CodeView 591
deffile field 586, 604

LINK *(continued)*
executable file
creating 570
field, described 586, 601
packing 592
size, minimizing with stub files 610
extended dictionary, overriding in search 595
far call 592–593, 595
fields
default responses 586
described 584
filename extensions 585
fix-ups
described 608–609
warning, issuing 600
groups (of segments) 608
incremental linking, preparation for 593
invoking 583–584
libraries field 586, 601
library
overriding default in search 595, 603
searching 602–603
line numbers in map file 594
linking
process, information display 594
steps described 606
mapfile field 586, 601
memory requirements 610
module-definition file 586, 604
object files field 600–601
options
abbreviations 587
conflicting 588
environment variable 587–588
field described 586–587
invalid for BASIC programs 590
numeric arguments 587
real/protected modes 529
valid for BASIC programs 588–589
viewing the options list 593
OS/2 programs 528–529
overlay
creating 571
described 612–614
interrupt number, setting 596
overview 570, 583–585
pausing during linking 598–599
prompts
described 604–605
response file 605–606

LINK *(continued)*
public (global) symbol, listing 595
Quick libraries, creating 570, 600, 620
response file
defined 583
described 605–606
search procedure 538–539
segment
address (offset/canonical frame number) 607
alignment types 607
code segment *See herein* code segment
combining 607–608
data segment *See herein* data segment
fix-ups 600, 608–609
frame number 607
groups 608
ordering (forcing) 591–592, 607
ordering (without null bytes) 596
setting maximum number 600
sign-on logo, suppressing 596
stand-alone library, creating 565
stub files 610–612
syntax 584
terminating LINK operation 583
unresolved references, fixing 608–609
window, specifying type 599
Linking
See also LINK
code with far strings 413
creating overlays 571
creating Quick libraries 570–571
LINK steps described 606
OS/2 programs 528–529
pausing 598
preparation for ILINK 593
process, display of information about 594
search procedure of LINK 538–539
with language libraries 428
with overlays 612–613
with stub file 571, 610–612
Listing file 556, 558
Literal constant 707
LOC function
compared to SEEK function 111
device compatibility 114, 116
Local constant 723
Local symbolic constant 725
Local variable
automatic/static variables 65–67
described 723–725

Local variable *(continued)*
FUNCTION procedure 38
SUB procedure 36
LOCATE statement
positioning, changing shape of text cursor 88–89
text coordinates, compared to graphics coordinates 150
LOF function 114, 116, 520
Logical operations 732, 736
Logical operators
Boolean expression 4
described 735–737
type conversion 729
values returned 736
LONG data type (ISAM) 333
Long integer
data type
described 702
suffix 695
numeric constant 709
Loop
See also Looping structures
condition, defined 16
defined 16
error occurring in 280
executing
indeterminate number of times 23–27
specific number of times 16–21
repeating as long as condition is true 21–23
speed considerations 549
suspending program execution 21
Looping structures
See also Loop
defined 16
DO...LOOP loop 23–29
FOR...NEXT loop 16–21
WHILE...WEND loop 21–23
Low level initialization error message 413
Lowercase letter
See also Case sensitivity/insensitivity
converting to uppercase 143
/Lp option
BC 559
protected-mode object file 528
LPT ports 114
/Lr option
BC 560
real-mode object file 528
routine to be used in Quick library 617
LSET function 105–106, 136
.LST filename extension 556, 558
LTRIM$ function 137, 141

M

/M option
ISAMCVT utility 391
LINK 589, 595
Macro
NMAKE macros *See* NMAKE
used with DRAW statement 190–192
Main module *See* Module
MAKE, compared to NMAKE 658–659
Make EXE File command (QBX)
See also /D option
modules 272
OS/2 program 527
Make Library command (QBX) 274, 619
Managing projects and files *See* NMAKE
MANDEL.BAS 211–217
Mandelbrot Set, sample program 211–217
Map file (LINK) 601
.MAP filename extension (LINK) 585
/MAP option (BUILDRTM) 664
MASM
See also Assembly language
calling BASIC 498–500
MATB.BAS 568
Math
arithmetic operations 733–734
floating-point math operations 559, 561–562
Matrix toolbox file 568
/MBF option (BC) 560
MCGA 223
Memory
expanded *See* Expanded Memory Specification
how BASIC uses memory 531–537, 719
ISAM memory management *See* ISAM
LINK, requirements 610
management *See* Ch. 3 in the *Getting Started* booklet
overlays, reducing program memory requirements 745
Quick library, determining available memory 628
references in protected mode 526–527
unused space, return 407
variable types, storage 717
video memory pages 204

Menu library 624–625
Menu toolbox file 568
MENU.BAS 568, 625
Message *See specific message*
Microsoft Binary format, converting functions to 560
Microsoft Help File Creation Utility *See* HELPMAKE
Microsoft Library Manager (LIB) *See* LIB
Microsoft Program Maintenance Utility *See* NMAKE
Microsoft QuickPascal 418
Microsoft Segmented-Executable Linker *See* LINK
MID$ function 141–142
MID$ statement 144
Minus sign (-)
  See also Dash (-)
  delete-command symbol (LIB) 579
  special character set 696
Mixed-language programming
  address, passing 471
  ALIAS keyword 430
  arguments, passing 451–454
  array
    declaring 467
    indexing 466–467
    more than two dimensions 469–470
    overview 463–464
    passing between modules 468–469
    passing from BASIC 464–466
    row-major or column-major storage 467–468
    storage, comparing BC to QBX 464
  assembly language *See* Assembly language
  /Ax option 427
  B_OnExit routine 474–475
  BASIC *See* BASIC
  BYVAL keyword 427, 430–431, 451
  C *See* C (language)
  calls
    See also specific language
    calling conventions 423–425, 450
    default conventions for each language 450
    interfaces 429–432, 443–444
    overview 419–420
    routine, language equivalents table 420
  CALLS statement 432, 451–452
Mixed-language programming *(continued)*
  CDECL keyword
    calling assembly language routine 486
    described 429–430
  CLEAR statement 440
  common block, passing 472
  compiling 427–428
  data type
    declaration characters 430, 431
    described 470–471
  debugging at source level 418
  DECLARE statement 429–430
  error handling 442–443
  event handling 442–443
  extern procedure 443
  external data defined 471
  far keyword 427–428, 444–445
  far strings *See* Far string
  fastcall keyword 425
  file I/O 442
  FORTRAN *See* FORTRAN
  fortran keyword 421, 431–432, 443–445
  function calls 419–420
  /Gc option 421, 431
  GetSpace$ procedure 441–442
  insufficient heap space error message 440
  INTERFACE statement 443
  linking with language libraries 428
  malloc 440
  MASM *See* Assembly language; MASM
  memory allocation 440
  naming conventions 420–423, 450
  near keyword 428, 445
  near strings 417
  nmalloc 440
  numeric data 434–436
  overview 417–418
  parameter
    order on stack 424–425
    passing 423, 425–427
    using a varying number of 473–474
  parameter list 430–431
  Pascal *See* Pascal
  pascal keyword 431–432, 443–445
  passing by reference 451
  passing by value 431, 451
  procedure calls 419–420
  Quick library, creating 622–623
  QuickPascal, incompatibility 418

Mixed-language programming *(continued)*
record data type (FORTRAN, Pascal) 470–471
restrictions 418
SEG keyword 431, 432
SETMEM function 440
stack, order of parameters 424–425
string
formats 456–458
passing between languages 459–463
space, allocating 441–442
struct data type (C) 470–471
structured data types defined 463
type-declaration characters 430, 431
user-defined data type (BASIC) 470–471
MKD$ function, converting to Microsoft Binary format 560
MKS$ function, converting to Microsoft Binary format 560
MOD operator 733
Modify EXE File command (QBX) 558
Module
compiling 272
custom run-time *See* Custom run-time module
DECLARE statement, including 271
defined 267
described 741
event trapping across modules 315–316
loading 270–271
main module 268–269, 741
moving procedures 270
non-main module 270
overview 267, 741
procedure-level module 270
programming tips 272, 274
Quick library
creating 274
relationship to modules 273, 617–618
run-time *See* Run-time module
variables, sharing 271–272
Module-definition file (LINK) 604
Module-level code
defined 268, 741
error handling *See* Error handling
event-handling routine, location in 303
use suggestions 267
Modulo arithmetic 733–734
Mouse
OS/2 include file 523
Quick library routine 624–625
toolbox file 568
MOUSE.BAS 568
MOVE keyword 520
MS-DOS support for ISAM 319
MS/ISAM file conversion 391–393
Multiple threads 526
Multiplication symbol (*) 696
Music
event, trapping 303, 308–309, 560
statements, unavailable in protected mode 527

N

/N option (NMAKE) 633
Naming conventions
files *See* Filename
ISAM 375
mixed-language programming 420–423, 450
run-time modules 529
Near heap 531, 719
near keyword 428, 445
Near memory 403, 531, 719
Near string
in mixed-language programming 417
program speed considerations 547
storage
compared to far string 403
location in memory 719
NMAKE
arguments described 632
at sign (@), command modifier 638
colons (::) as separator character 639
command
description block component 636
modifiers 638–639
preventing display during execution 633, 638
command file 634–635
#comment description block component 636
compared to MAKE 658–659
customizing 654
dash (-), command modifier 638
date of modification 632–633
defined 631
dependency line defined 635

NMAKE *(continued)*
dependent
defined 631, 635
description block component 636
description block
components 635–636
described 640
specifying more than one for same target 639
description file
command, displaying without executing 633
escape character 637
overview 635–636
specifying 633
directives 650–652
error checking, turning off 638
error output, sending to file/device 633
escape character, description file 637
exclamation point (!), command modifier 639
exit codes, ignoring 633
filename argument 632
help information 633
in-line file 654–656
inference rules
described 647–650
ignoring predefined and TOOLS.INI inference rules 633
invoking NMAKE
by using a file 634–635
from command line 631–632
macro
defining 641
definition *See herein* macro definition
ignoring predefined and TOOLS.INI macros 633
inherited macros 647
overriding macros in description file 633
overview 640–641
predefined 643–646
substitution 642–643, 646
using 641–642
macro definition
canceling 642
form 641
order of precedence 647
printing 633
macrodefinitions argument 632
MAKEFILE as description file 633, 634
message suppression
copyright 633
error/warning 632
options
table of command-line options 632–633
turning on/off with directive 651
options argument 632
overview 631
pseudotargets 652–654
response file, generating 654
sequence of operations 657–658
status code 633
target
building all targets option 632
defined 631
description block component 636
description, printing 633
multiple description blocks 638
primary, defined 640
pseudotargets 652–654
specifying in multiple description blocks 639–640
status code 633
target...argument 632
TOOLS.INI file 654
wildcard characters 637
No more virtual memory error message (LIB) 576
/NOD option
LIB 576
LINK 589, 595
/NOE option
LINK 589, 595
linking with stub files 539
/NOF option (LINK) 589, 595
/Nofrills option *See* Ch. 3 in the *Getting Started* booklet
/NOI option
LIB 576
LINK 589, 596
/NOL option (LINK) 589, 596
/NOLOGO option
LIB 576
NMAKE 601
/NON option (LINK) 589, 596
Nonexecutable statement defined 698
/NOP option (LINK) 589, 596
Not equal (< >), relational operator 4, 137, 734
NOT operator 4–6, 736

Number
  complex, described 211
  converting binary to hexadecimal 183
  displaying on screen with PRINT statement 78–80
  frame number (LINK) 607
  in filenames 93
  storage, comparing random-access to sequential file 103
  string representation of 143–144
Number sign (#)
  comment character
    export list for custom run-time module 662
    NMAKE 636
  double-precision type-declaration character 695, 702, 715
Numeric array
  dynamic *See* Dynamic array
  static *See* Static array
  storage 535–536, 720
Numeric character set 695
Numeric constant 708–712, 728
Numeric data type
  See also specific data type
  described 701–707
  equivalents table 454–456
Numeric variable
  assignment 713
  memory location 534, 719
  type-declaration 715–717
NumLock key, trapping 306

O

/O option
  BC 560
  executable file size considerations 544
  HELPMAKE 674, 676
  LINK 589, 596
.OBJ filename extension
  See also Object file
  BC 558
  LINK 585
Object-code library defined (LIB) 573
Object code, listing with /A option 558
Object file
  adding to library (LIB) 578–579
  creating (BC) 556
  creating Quick library 616, 622–623
Object file *(continued)*
  debug file for use with Microsoft CodeView 560
  line-number records 560
  linking (LINK) 600–601
  protected-mode, creating with /Lp option 559
  real-mode, creating with /Lr option 560
  special-purpose object files supplied with Microsoft BASIC *See* Stub files
Object module defined (LIB) 578
Object-module libraries
  creating Quick library from 623
  described 564–568
Octal number
  LINK argument 587
  numeric constant 708, 709
Olivetti Color Board 223
ON ERROR GOTO statement 276–281, 295
ON ERROR RESUME NEXT statement 293–294
ON ERROR statement
  compiling with /E or /X option 559, 560
  effect of zero (0) as line number 697
On *event* GOSUB statement 316
ON *event* statement 521, 697
ON...GOSUB statement 15
ON KEY GOSUB statement 299, 305–307
ON PLAY GOSUB statement 308
ON SIGNAL statement 522
ON UEVENT GOSUB statement 310
Online help
  creating/customizing *See* HELPMAKE
  for mixed-language programming (QBX) 518
OPEN COM statement 115–116
OPEN keyword 520
OPEN PIPE statement 522
OPEN statement
  adding data to sequential file 96
  binary-access file 110
  data file 91–93
  ISAM 335–336
  LEN = clause 104
Operations line, extending (LIB) 582
Operators
  arithmetic operators 733–734
  C unary/binary operators (NMAKE) 651
  functional operators 738
  logical 4, 729, 735–738
  order of precedence 731–732
  relational operators 4, 137–138, 734–735
  string operators 739

Optimizing
far call (LINK) 592
procedure calls 562
program size and speed, overview 531, 538
OPTION BASE statement
nonexecutable statement 698
not allowed in procedure definition 41
Options
BC 558–562
BUILDRTM 664
HELPMAKE 673–676
LIB 576
LINK
list 588–590
real- and protected-modes 529
viewing list on screen 593
NMAKE 632–634
OR logical operator 4, 736
OR option, with PUT graphics statement 196–198
OS/2 programming
compiling 527–528
debugging 529
editing source code 519
external run-time modules 527
include files 523–526
ISAM not supported 319
linking 528–529
overview 519
protected mode
calling OS/2 functions from 523–527
object file, creating 559
running BASIC programs under 530
statements and functions 520–522
real-mode programming 519
/Ot option 548, 560, 562
Out of memory error message 363, 560
Out of stack space error message 68
OUT statement, unavailable in protected mode 521
Output
See also I/O
displaying formatted output 80
redirecting standard output 78
OUTPUT mode 96–97
Overflow error message 729
Overlay
creating with LINK 571
interrupt number, setting 596
linking with 612–614
overview 745
program design 543
size and number supported 533
Overlay manager 614
Overwriting
data file when opening 91
variable values 67

P

/P option (NMAKE) 633
/PAC option (LINK) 589, 596–597
/PACKD option (LINK) 589, 597
PAINT statement
background$ argument 187
bordercolor& argument 178, 179, 187
pattern/tiling 179–189
repainting 187–188
solid colors 178–179
STEP keyword 152
Painting
See also Color; Drawing
pixels 148–150
repainting 187–188
with patterns (tiling) 179–190
with solid color 178–179
Palette *See* Presentation Graphics toolbox
PALETTE statement
changing colors 173, 175–176
unavailable in protected mode 521
PALETTE USING statement
changing colors 175–176
unavailable in protected mode 521
PALETTE.BAS 176–177
/PA:*number* option (LIB) 576, 577
Parameter
compared to argument 44–45
defined 44
list *See* Parameter list
passing
in mixed-language programming 425–427
speed considerations 548, 549
Parameter list
components 45
procedure definition 40
variable
checking with DECLARE statement 50–51
type-declaration methods 47
Parentheses (( ))
in Boolean expression 4–5
order of arithmetic operations 733
special character set 696

Pascal
array
declaring 467
indexing 466
calling BASIC 449–450
calling conventions 450–451
language equivalents for routine calls 420
naming conventions 420–423, 450
numeric data types, equivalents 454–455
passing arguments 454
passing parameters 427
passing strings to BASIC 463
record data type 470
string format 458
pascal keyword 421, 431–432, 443–445
Passing
addresses 471
arguments
See also Passing by reference; Passing by value
described 44–50
arrays 47–49, 55, 464–466, 468–469
constants 45–46
expressions 45–46
parameters 423–427, 548, 549
records 49–50
strings
between languages 459–463
far strings 408–410
fixed-length 517–518
in QBX 515
variable-length 490–492, 493–517
variables
described 47–50
in chain 70–71
Passing by reference 55, 451–454, 743–744
Passing by value 56–57, 431, 451–454, 549, 743–744
Pattern
editing tiles, sample program 217–222
painting with 179–190
/PAU option (LINK) 589, 598–599
Pausing program execution 21
PEEK function 521, 526
PEEK statement 404–406
Pel *See* Pixel
PEN function, unavailable in protected mode 522
PEN statement
event trapping 303, 560
unavailable in protected mode 522
Percent sign (%), integer type-declaration character 695, 702, 715
Period (.), special character set 696
Peripheral *See* Device
Perpetual calendar, sample program 116–121
PGBAR.BAS 237–239
PGPIE.BAS 233–236
PGSCAT.BAS 242–244
Physical coordinates
defined 166
translating to window coordinates with PMAP function 170–171
Picture element *See* Pixel
Pie chart *See* Presentation Graphics toolbox
Pie shape/wedge, drawing 161
Pipes and queues, OS/2 include file 523
Pixel
aspect ratio 162
bit planes in EGA and VGA screen modes 189
bits per pixel, computing 180
defined 148
locating with coordinates 149–150
plotting with PSET, PRESET statements 150–151
PLAY "MB" statement 308
PLAY ON statement 308
PLAY statement
trapping music 303, 308–309, 560
unavailable in protected mode 522, 527
PLOTTER.BAS 190–192
Plus sign (+)
add-command symbol (LIB) 578
special character set 696
string concatenation operator 136–137, 739
PMAP function
translating physical and window coordinates 170–171
use of floating-point math package 542
/PM:*type* option (LINK) 589, 599
POINT function
graphics cursor location 170–171
use of floating-point math package 542
Pointer
far-string pointer 408–410
file pointer 110–111
POKE statement
direct far-string processing 404
in protected mode 522, 526
Polling, compared to event trapping 298–299
POS(*n*) function 89–90
.PRECIOUS: pseudotarget (NMAKE) 653

Presentation Graphics toolbox
adapters supported 223
analysis routines 262
axis
customizing 249–251
defined 229
AxisType 249–251
bar chart
defined 228
sample program 237–239
styles 230
border style
described 261
palette 253
categories defined 228
chart *See herein* bar chart; column chart; line chart; pie chart; scatter diagram
Chart routine 261
chart styles, table of 230
chart window
customizing 246–247
defined 230
ChartEnvironment
described 245, 253–253
user-defined types nested within 246
ChartErr 233
ChartMS routine 261, 262
ChartPie routine 261
ChartScatter routine 261
ChartScatterMS routine 261, 262
ChartScreen routine 226, 261
ChartStyle variable 256
ChartType variable 253
ChartWindow variable 256
color
axis, grid lines 250
border frame of window 247
described 257–258
in chart environment 245
palette 253
text in legend window 252
title 248
column chart
defined 228
examples 225–227, 239–241
styles 230
customizing, described 244
data point defined 227
data series defined 227

Presentation Graphics toolbox *(continued)*
data window
customizing 246–247
defined 230
DataFont variable 256
DataWindow variable 256
default values, modifying 245
DefaultChart routine 226, 261
Env variable 232
error handling 233, 236
files supplied with BASIC 224, 568
fill pattern
described 259–260
palette 253
font
graphics 224, 266
text 252
grid lines 249
labelling routines 262
legend
customizing 252–253
defined 231
Legend variable 256
LegendType 252–253
line chart
defined 229
example 239–240
multi-series, example 263–264
styles 230
line styles
described 259
palette 253
MainTitle variable 256
non-numeric data *See herein* categories
numeric data *See herein* values
overview 223–224
palettes 253–257
PGBAR.BAS 237–239
PGPIE.BAS 233–236
PGSCAT.BAS 242–244
pie chart
defined 228
exploded, defined 228
sample program 232–236
styles 230
plot character
described 261
palette 253
RegionType 246–247

Presentation Graphics toolbox *(continued)*
routines, table of 261–262
scatter diagram
defined 229
program example 241–244
styles 230
SubTitle variable 256
tick marks 251
TitleType 248
user-defined data types 244–246
values defined 228
window
See also herein chart window; data window; legend
elements 246–247
x-axis
See also herein axis
defined 229
XAxis variable 256
y-axis
See also herein axis
defined 229
YAxis variable 256
Presentation manager applications 599
PRESET option, with PUT graphics statement 196–198
PRESET statement
color argument 172
plotting points 150–151
STEP keyword 152
PRINT statement 78–80
PRINT # statement 98–100
PRINT USING statement
displaying formatted output 80
use of floating-point math package 542
Print zone 79, 99
Printer devices supported 114
Printing to screen *See* Screen
Procedure
benefits of using procedures 35
calling
before defined (forward reference) 51, 52
FUNCTION procedure 42–43
procedure calling itself (recursion) 67–69, 744
SUB procedure 43–44
comparing parameters and arguments 44–45
comparing procedures with subroutines 36–39
DECLARE statement 50–55
defined 35
Procedure *(continued)*
definition *See* Procedure definition
ending/exiting 41
external procedure defined 53
forward reference defined 51
FUNCTION *See* FUNCTION procedure
in modules 267–270
in Quick library 274, 617
minimizing generated code in programming 543
optimizing execution speed with /Ot option 548, 562
overview 35–36, 741–745
passing arguments *See* Passing
recursive procedure 67–69, 744
sharing variables
across modules 62–64
described 57–58
with all procedures in module 60–62
with specific procedures in module 58–59
SUB *See* SUB procedure
subprogram defined 36
subroutine defined 36
type checking of arguments 50–55
variable
aliases 64–65
automatic/static 65–67
sharing *See herein* sharing variables
Procedure definition
certain BASIC statements not allowed 41
comparing parameters and arguments 44–45
module, use of include file 53–55
nesting prohibited 41
not allowed in include file (QBX) 55
syntax 40–41
Procedure-level error handling 277, 281–284
Process control, OS/2 include file 523
Program
chaining 69–71
compiling *See* Compiling
data
managing for speed 546–547
saving space 537–538
entry point (main module) 741
examples *See* Sample programs
in other languages *See* Mixed-language programming
maintenance utility *See* NMAKE
minimizing generated code 542–543
modules *See* Module

Program *(continued)*
OS/2 *See* OS/2 programming
overlays *See* Overlays
procedures *See* Procedure
protected mode *See* OS/2 programming
real mode *See* Real-mode program
run-time routine 538, 541
sample *See* Sample programs
size, optimizing 531, 538–544
speed, optimizing 531, 544–551
splitting with CHAIN statement 69
statements *See* Statement
storage in memory, described 531–537
subprogram defined 36
suspending execution 21
terminate-and-stay-resident *See* ISAM TSR
updating *See* NMAKE
Program line
BASIC syntax 696–699
QBX line length 699
Program Maintenance Utility *See* NMAKE
Programming
conventions used in this manual *xxxi-xxxii*
languages, combining *See* Mixed-language programming
with modules *See* Module
PROISAM.EXE 375–377
PROISAMD.EXE 375–377
Project management *See* NMAKE
Prompt
input prompts 85
LIB prompts 581–582
Protected-mode program *See* OS/2 programming
PSET option, with GET and PUT graphics statements 196–199
PSET statement
color argument 172
plotting points 150–151
STEP keyword 152
Pseudotarget (NMAKE) 652–654
Public symbol, listing (LINK) 595
PUT statement
binary-access file I/O 110–111
graphics 195–199
random-access file I/O 109
STEP keyword 152

Q

/Q option
LINK 589, 600
NMAKE 633
QBX
array data storage compared to BC 464
compiling
BC options available 561
far string storage as default 404, 535, 720
huge arrays (/Ah option) 537
modules 272
OS/2 program 527–528
DECLARE statement, generating 35, 50–51
/E option *See* Ch. 3 in the *Getting Started* booklet
/Ea option *See* Ch. 3 in the *Getting Started* booklet
/Es option *See* Ch. 3 in the *Getting Started* booklet
invoking LINK 583
line length limitations 699
linking OS/2 programs 528–529
loading
modules 270
Quick library 273, 625–626
memory management *See* Ch. 3 in the *Getting Started* booklet
modifying BC command options 558
online Help for mixed-language programming 518
passing strings 515
Presentation Graphics programs 223
Quick library, creating from 618–620
relational operations, differing results with .EXE file 735
starting ISAM for use in 375–377
QBX.QLB, Quick library supplied with Microsoft BASIC 615
.QLB filename extension
See also Quick library
described 564, 617
LINK executable file 585
QLBDUMP.BAS 111–113, 626
Question mark (?)
special character set 696
wildcard character (NMAKE) 637

Queues, OS/2 include file 523
Quick BASIC Extended development environment *See* QBX
Quick library
advantages of using 568–569
B_OnExit routine 627–628
compared to user library from QuickBASIC 569
creating
files required 616
from BASIC source modules 621–622
from command line 620
from .LIB files 623
from other language modules 622–623
from other Quick library 623–624
from QBX 618
overview 274, 569, 617
source files, types 616
with LINK 600
described 568–569
event trapping 620
filename extension (.QLB) 564, 617
in expanded memory *See* Ch. 3 in the *Getting Started* booklet
/L option 615, 616, 618, 625
.LIB file 620
loading (QBX) 273, 625–626
.MAK file, updating 626
menu routine library 624–625
mouse routine library 624–625
.OBJ file 620
object file (.OBJ), deleting 620
object-module library (.LIB) 620, 629
overview 273, 615
procedure declarations in 51, 55
programming considerations 627–629
QBX.LIB for use outside of QBX environment 615
QBX.QLB, supplied library 615
.QLB filename extension 564, 617
QLBDUMP.BAS utility 111–113, 626
relationship to modules 273
routines, creating from BASIC toolbox files 624–625
/RUN option 626
search path (QBX) 626
size, determining maximum available 628
suggestions for use 273
Quick library *(continued)*
supplied library (QBX.QLB) 615
termination routines 627–628
viewing contents with QLBDUMP.BAS 626
window routine library 624–625
QuickHelp file format *See* Help database; Help text file
QuickPascal modules, not compatible with other languages 418
Quotation mark, single (') *See* Apostrophe
Quotation marks, double ("), field delimiter 96

R

/R option
NMAKE 633
row-major array storage order (BC) 467, 560
Radian, unit of angle measure 158–160
Radius defined 156
RAM *See* Random-access memory
RAMDRIVE.SYS device driver *See* Ch. 3 in the *Getting Started* booklet
Random-access file
compared to
binary access 110–111
ISAM file 322
sequential file 91
defined 91
entering data 103, 105–106
indexing, sample program 121–130
record
appending 107
calculating number of 107
defining 103–104
length 104–105
reading 108–110
storing 103, 105
writing to 109
Random-access memory (RAM)
See also Memory
how BASIC uses memory 531–537, 719
RANDOM function, use of floating-point math package 542
READ function, use of floating-point math package 542
Real-mode object file 560

Real-mode program
compiling an OS/2 program 527–528
creating 519
LINK options 529
Record
defined 49, 90, 712
in ISAM file *See* ISAM
in random-access file *See* Random-access file
in sequential file *See* Sequential file
passing to procedure 49–50
variable, declaring 715
Rectangle, drawing *See* Box
Recursive procedure
comparing FUNCTION and DEF FN 39
described 67–69, 744
searching a directory, sample program 72–75
REDIM statement
declaring global variable 722
fixed-length string 135–136
SHARED attribute 60–62
Redirecting standard output 78
Redo from start error message 85
Relational operators
comparing strings 137–138
described 734–735
table of 4
REM statement 698
Repeating statements *See* Loop; Looping structures
Response file
defined 573
LIB 582–583
LINK 605
NMAKE 654
RESUME NEXT statement
exiting error-handling routine 278–279
program continuation after error-handling routine 287
RESUME statement
exiting error-handling routine 278–279
program continuation after error-handling routine 287
zero (0) as line number, effect 697
RETRIEVE statement (ISAM) 342, 520
Rich Text Format (RTF) 671, 687–688
RIGHT$ function 141
ROLLBACK statement
ISAM 382, 383
protected mode 520
Row
changing number of rows 81
screen rows described 78
skipping spaces in printed output 80–81
RSET function 105, 136
RTF file format 671, 687–688
RTRIM$ function 137, 140–141
Run Make EXE File command *See* Make EXE File command
Run Make Library command *See* Make Library command
/RUN option 626
Run-time error messages (ISAM) 399–401
Run-time library
created by BUILDRTM utility 666
described 566–567
naming conventions for different operating modes 529, 566, 665
substituting stand-alone library as default 560
Run-time module
customizing *See* Custom run-time module
default, creating with BUILDRTM utility 665
extended, creating 527
naming conventions for different operating modes 529, 566, 665
Run-time routines, controlling program size 538, 541

S

/S option
BC 560
HELPMAKE 674
NMAKE 633
SADD function 408–409
Sample code, copying to file or executing 518
Sample programs
animation by rotating colors (PALETTE.BAS) 176–177
ball, bouncing (BALLPSET.BAS) 199–202
ball, bouncing (BALLXOR.BAS) 202–203
bar chart (PGBAR.BAS) 237–239
bar-graph generator (BAR.BAS) 205–210
blanks, replacing with tab characters (ENTAB.BAS) 721–722
calendar, perpetual (CAL.BAS) 116–121
carriage-return and line-feed filter (CRLF.BAS) 31–34
checkbook balancing (CHECK.BAS) 29–31

Sample programs *(continued)*
color palette combinations (COLORS.BAS) 174–175
cube, rotating (CUBE.BAS) 204–205
drawing, graphics macros (PLOTTER.BAS) 190–192
indexing a random-access file (INDEX.BAS) 121–130
ISAM database (BOOKLOOK.BAS) 342, 345–354
Mandelbrot Set (MANDEL.BAS) 211–217
pattern tile editor (EDPAT.BAS) 217–222
pie chart (PGPIE.BAS) 233–236
recursive directory search (WHEREIS.BAS) 72–75
scatter diagram (PGSCAT.BAS) 242–244
sine-wave function, graphing (SINEWAVE.BAS) 168–169
single-precision values, internal format (FLPT.BAS) 703–705
string, converting to number (STRTONUM.BAS) 145–146
terminal emulator (TERMINAL.BAS) 130–132
variables, automatic and static (TOKEN.BAS) 726–727
SAVEPOINT statement
ISAM 382, 383–384
protected mode 520
Scatter diagram *See* Presentation Graphics toolbox
Screen
colors *See* Color
column, described 78, 80–81
configuration, standard 78
coordinates
See also Coordinates
overview 148–150
graphics viewport 164–166
line length, wrapping 79
modes *See* Screen mode
page *See* Screen page
pixels *See* Pixel
printing output to screen
formatted output 80
speed considerations 550
with PRINT statement 78–80
resolution 149
rows, described 78, 80–81
text viewport 81–83
Screen mode
devices supported for output 113–114
graphics output modes
bit planes 188–189, 193
color/monochrome display 172
selecting 148
Screen page
active page defined 204
graphics animation technique 204–205
visible page defined 204
SCREEN statement
Presentation Graphics 226
protected mode 522
selecting screen mode 148–149
selecting visual/active screen page 204–205
using constant to control program size 507
SCRN: device 114
Scrolling text in viewport 82
/SE option (LINK) 589, 600
SEEK function 111
SEEK statement 91, 111
SEEKEQ statement (ISAM) 356
SEEKGE statement (ISAM) 356
SEEKGT statement (ISAM) 356
SEG keyword 431, 452
Segment *See* LINK
Segmented-Executable Linker *See* LINK
SELECT CASE statement
compared to block IF...THEN...ELSE statement 10–11
compared to ON...GOSUB statement 15–16
described 10–15
use for structured decisions 8
Semaphores, OS/2 include file 523
Semicolon (;)
LIB syntax 574
library consistency check 575
LINK syntax 584
special character set 696
suppressing carriage-return line-feed sequence
in INPUT statement 86
in PRINT statement 79
Sequential file
access, compared to ISAM 322
adding data 96–97
appending data 92, 98
compared to random-access file 91
defined 91
I/O speed considerations 550

Sequential file *(continued)*
jumping to specific byte 91
reading data 97–98, 100–102
record, storing 95–96
writing to file with PRINT # statement 98–100
Serial port 113, 115–116
Session manager, OS/2 include file 523
SET (DOS command) 538
SETINDEX statement
ISAM 337–338, 343
protected mode 520
SETMEM function 522
Setup *See* Ch. 3 in the *Getting Started* booklet
SHARED attribute, sharing variables globally 60–62, 722
SHARED statement
declaring fixed-length string 135–136
nonexecutable statement 698
overridden by STATIC statement 66
sharing variables with specific procedures 58–59, 724
Sharing variables
across modules 62–64
aliases problems 64–65
described 57–58
with all procedures 60–62
with specific procedures 58–59, 724
SHELL statement 522
Shift key combination, trapping 305–307
Sign-on logo, suppressing (LINK) 596
Signals, OS/2 include file 523
.SILENT: pseudotarget (NMAKE) 653
SINEWAVE.BAS program 168–169
Single-precision numeric constant 710, 711
Single-precision numeric data type 103, 695, 702
Slash (/)
floating-point division symbol 733
option indicator (BC) 558
special character set 696
SLEEP statement
alternative to suspending with FOR...NEXT loop 21
holding output on screen 153
SMARTDRV.SYS device driver *See* Ch. 3 in the *Getting Started* booklet
Soft keys *See* Function keys
SOUND statement
unavailable in protected mode 522, 527
use of floating-point math package 542
Space
blank spaces
effect of leading/trailing 138
generating a string of 143
trimming 140, 141
field delimiter 96
leading/trailing spaces
removing from string 137
significance in string comparison 138
skipping spaces in row of output 80–81
special character set 695
SPACE$ function 143
SPC statement 80–81
Special character
in filenames 93
set, recognized by BASIC 695–696
Speed of program
compiling programs 544–545
hints for program execution 551
managing data 546–547
optimizing control-flow structures 547–550
optimizing I/O 550
SSEG function 404, 408
SSEGADD function 408, 492
/ST option (LINK) 590
Stack
frame, speed considerations 549
order in which arguments are pushed 424–425
size, adjusting 68–69
storage in near memory 531, 533
STACK function 68–69
STACK statement 68–69
Stand-alone library *See* Library
Standard input
See also Input; I/O
defined 84
Standard output, redirecting 78
Statement
See also specific statement
changes for use in protected mode 520–522
executable, defined 698
execution order 3
nonexecutable, defined 698
repeating *See* Loop; Looping structures
Statement block defined 3
Static array
compared to dynamic array 547
described 725–726
numeric 534, 536, 719
string 535, 720
Static array data type (ISAM) 334

STATIC attribute 40, 65
$STATIC metacommand 726
STATIC statement
declaring fixed-length string 135–136
declaring local variable 38, 66–67, 723
nonexecutable statement 698
Static variable 65–67, 726–727
STEP keyword
in FOR...NEXT loop 17
making coordinates relative 152
not allowed with VIEW statement 164
STICK function, unavailable in protected mode 522
STOP statement 292
STR$ function 144
STRIG function, unavailable in protected mode 522
STRIG statement
event trapping 303, 560
unavailable in protected mode 522
String
alphabetizing 138
array *See* String array
blank spaces
effect of leading/trailing 138
generating 143
trimming 140, 141
characters
allowed 133
changing case of letters 143
repeating 142–143
retrieving from any part of string 140–142
stored as ASCII code 134
combining 136–137, 739
comparing 4, 137–138, 739
concatenation *See herein* combining
constant *See* String constant
data handling in mixed-language programming 490–493
defined 133
descriptors 441
expression *See* String expression
far *See* Far string
fixed length *See* Fixed-length string
formats
BASIC 456–457
C 457–458
FORTRAN 458
Pascal 458
near string *See* Near string
String *(continued)*
numeric representation of a string 143–146
operators 739
passing
between languages 459–463
in QBX 515
processing routines 490
searching for string inside another string 138–139
sorting 137–138
space, allocating 441–442
storage in memory 531
transferring string data between languages 490–492
variable *See* String variable
variable length *See* Variable-length string
writing to object file with /S option 560
String array
dynamic
See also Dynamic array
defined 535, 720
static
See also Static array
defined 535, 720
storage in memory 535, 720
variable-length, passing 515–517
String constant 133–134, 708
String data type
dollar sign ($) as suffix 695
ISAM 334
String expression
defined 133–134, 739
storing in record 99
STRING$ function 142–143
String operators 739
String space corrupt error message 406
String variable
declaring 133–134, 715
type mismatch 728
StringAddress routine 490, 492
StringAssign routine 490–492
StringLength routine 490, 493
StringRelease routine 490, 492
STRTONUM.BAS 145–146
Stub file
See Ch. 3 in the *Getting Started* booklet
described 538–539
in custom run-time module 663
included with Microsoft BASIC 540–541, 611–612
linking with 571, 610

SUB...END SUB procedure *See* SUB procedure
SUB...END SUB statement 41
SUB procedure
  calling 37, 43–44
  compared to GOSUB subroutine 36–38
  DECLARE statement
    generating in QBX 51
    syntax in mixed-language programming 429–430
  definition 40–41
  described 742–743
  improving speed of calls 548–549, 562
  in multiple modules 37
  listing in custom run-time module 663
  recursive 744
  saving data space by using local variables 537
  STATIC attribute 40
  syntax 40–41
  variables
    name restriction 40
    local by default 36
Subprogram
  See also SUB procedure
  defined 36
Subroutine
  compared with procedures and functions 36–39
  defined 36
Subscript out of range error message 537
.SUFFIXES:*list* pseudotarget (NMAKE) 653
Support toolbox file 568
Suspending program execution 21
Switches *See* Options
.SYM file (LINK) 593
Symbol defined more than once error message 418
Symbol, listing public/global (LINK) 595
Symbol multiply defined error message (LIB) 576
Symbolic constant
  described 712
  global 722
  scope 721, 725
  string 134
Syntax notation conventions used in manual *xxix–xxxii*
System requirements *See* Ch. 2 in the *Getting Started* booklet

T

/T option
  BC 560
  HELPMAKE 675
  LINK 590
  NMAKE 633
TAB statement 81
Table, ISAM *See* ISAM
Terminal emulator, sample program 130–132
TERMINAL.BAS 130–132
Terminate and stay resident *See* ISAM TSR
Text
  changing in description file or macro (NMAKE) 640–643
  printing on screen 78–80
  scrolling 82
Text cursor 87–89
Text file
  See also Sequential file
  carriage-return line-feed filter 31–34
  Help *See* Help text file
Text viewport 81–83
TEXTCOMP function (ISAM) 355–356
Tile editing, sample program 217–222
Tiling 179–190
Time functions, add-on libraries 567
TIMER function, use of floating-point math package 542
TIMER statement, event trapping 303, 560
TOKEN.BAS 726–727
Toolbox files 568
TOOLS.INI file, customizing NMAKE 654
Transaction processing in ISAM 382–384
Trapping
  errors *See* Error trapping
  events *See* Event trapping
True expression 4–6, 734, 736
TSR *See* ISAM TSR
Type-declaration characters 695–696, 702, 715
TYPE...END TYPE statement
  accessing variables from two or more modules 271
  defining record in random-access file 104
  ISAM 320–321, 333
  nonexecutable statement 698
  not allowed in procedure definition 41
  user-defined data types 706

TYPE keyword 520
Type mismatch error message 143, 728
TYPE statement 706
Typographic conventions used in manual *xxix–xxxi*

U

UBOUND function 49
UCASE$ function 143
UEVENT ON statement 310–312
UEVENT statement, event trapping 303, 560
Unanticipated error *See* Error-handling
Underscore (_)
    not recognized as line continuation character (QBX) 699
    special character set 696
Unrecognized option name error message (LINK) 587
UNTIL keyword 28
Up arrow (^)
    See also Caret
    exponentiation symbol 696
    special character set 696
UPDATE statement (ISAM) 342, 520
Updating applications *See* NMAKE
Uppercase letter
    See also Case sensitivity/insensitivity
    converting to lowercase 143
User-defined data type
    described 706
    ISAM 334
    passing in mixed-language programming 470–471
User-defined event, trapping 303, 310–312, 560
User-defined key, trapping 304, 305–306

V

/V option
    BC 316, 560
    HELPMAKE 675, 676
VAL function
    numeric representation of string 144
    use of floating-point math package 542
Value
    assigning to variable 713
    comparing with relational operators 734
    constant *See* Constant
Value *(continued)*
    defined (Presentation Graphics toolbox) 228
    passing arguments by value *See* Passing by value
    returning true or false 4–6, 736
    unsigned (OS/2) 523
Variable
    aliases 64–65
    array variable
        declaring 717–718
        defined 712
    assigning values 713
    automatic variable 65, 726–727
    declaring
        AS clause 47, 715–716
        DEF*type* statements 716–717
        outside of QBX 61
        suffixes 715
    defined 712
    global *See* Global variable
    in input list 84–85
    local *See* Local variable
    naming 40, 714
    numeric *See* Numeric variable
    passing
        by reference 55–56
        by value 56–57
        in chained programs 70
        to procedure 47–50
    scope 721–725
    sharing 57–65, 724
    simple variable defined 712
    static variable 65–67, 726–727
    storage in memory 531, 533
    string *See* String variable
    type declaration 47, 715–717
Variable-length string
    arrays, passing 515–517
    comparing with fixed-length string
        using RTRIM$ function 140–141
        with relational operators 137–138
    declaring 715–716
    described 135
    passing strings
        BASIC and C 501–505
        BASIC and FORTRAN 505–510
        BASIC and MASM 493–500
        BASIC and Pascal 510–514

Variable-length string *(continued)*
passing strings *(continued)*
procedure 493
QBX 515
processing for mixed-language programming 489–493
storage in memory 534, 719
stored outside of DGROUP *See* Far string
VARPTR function, in protected mode 526
VARSEG function, in protected mode 522, 526
VGA *See* Video Graphics Array
Video Graphics Array (VGA)
color changes with PALETTE statement 173, 175
for graphics programs 147
Presentation Graphics toolbox 223
VIEW PRINT statement 81–82
VIEW SCREEN statement 165
VIEW statement
color argument 172
compared to VIEW SCREEN statement 165
syntax 164–165
Viewport
graphics viewport 164–166
text viewport 81–83
Virtual addressing defined 532
Visible page defined 204

W

/W option
BC 316, 560
HELPMAKE 675
LINK 589, 600
WAIT statement, unavailable in protected mode 522
Warnings from compiler, suppressing 560
Wedge/pie shape, drawing 161
WHEREIS.BAS 72–75
WHILE...WEND statement 21–23
WIDTH statement 81
Wildcard character (NMAKE) 637
Window
chart window *See* Presentation Graphics toolbox
coordinates, translating to physical coordinates 170–171
Window *(continued)*
data window *See* Presentation Graphics toolbox
specifying Presentation Manager type (LINK) 599
Window coordinates defined 167
Window library 624–625
WINDOW statement
changing coordinate system 166–167
compared to WINDOW SCREEN statement 167–168
order of coordinate pairs 171
use of floating-point math package 542
Window toolbox file 568
WINDOW.BAS 568, 625
WRITE # statement
compared to PRINT # statement 98–100
writing records to sequential file 96–97

X

x-axis defined 229
/X option
BC 560
error handling 299
NMAKE 633
XOR logical operator 4, 736
XOR option, with GET and PUT graphics statements 197–199

Y

y-axis defined 229

Z

/Z option 560
/Zd option 560
Zero (0)
as line number 697
in error-handling routine 277
operations which produce run-time errors 734
/Zi option 560

Microsoft®
Making it all make sense™

1089 Part No. 06824